

NAVAL POSTGRADUATE SCHOOL
Monterey, California

AD-A245 389



DTIC
ELECTE
FEB 04 1992
S B D

THESIS

VHDL BEHAVIORAL DESCRIPTION
OF DISCRETE COSINE TRANSFORM
IN IMAGE COMPRESSION

by

DENG, AN-TE

September 1991

Thesis Advisor:

Chin-Hwa Lee

Approved for public release; distribution is unlimited

92-02637



Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE			
1a REPORT SECURITY CLASSIFICATION Unclassified		1b RESTRICTIVE MARKINGS	
2a SECURITY CLASSIFICATION AUTHORITY		3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited	
2b DECLASSIFICATION/DOWNGRADING SCHEDULE			
4 PERFORMING ORGANIZATION REPORT NUMBER(S)		5 MONITORING ORGANIZATION REPORT NUMBER(S)	
6a NAME OF PERFORMING ORGANIZATION Naval Postgraduate School	6b OFFICE SYMBOL (If applicable) EW	7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School	
6c ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000		7b ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000	
8a NAME OF FUNDING/SPONSORING ORGANIZATION	8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c ADDRESS (City, State, and ZIP Code)		10 SOURCE OF FUNDING NUMBERS	
		Program Element No.	Project No.
		Task No.	Work Unit/Accession Number
11 TITLE (Include Security Classification) VHDL Behavioral Description of Discrete Cosine Transform in Digital Image Compression			
12 PERSONAL AUTHOR(S) Deng, An-Te			
13a TYPE OF REPORT Master's Thesis	13b TIME COVERED From To	14 DATE OF REPORT (year, month, day) September 1991	15 PAGE COUNT 139
16 SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
17 COSATI CODES		18 SUBJECT TERMS (continue on reverse if necessary and identify by block number)	
FIELD	GROUP	SUBGROUP	
		Image Compression; Discrete Cosine Transform; VHSIC Hardware Description Language; Top-Down design;	
19 ABSTRACT (continue on reverse if necessary and identify by block number) This thesis describes a VHSIC Hardware Description Language (VHDL) simulation of a hardware 8 x 8 Discrete Cosine Transform (DCT) which can be applied to image compression. A Top-Down Design approach is taken in the study, a discussion of DCT theory is presented, along with a description of the 1-D DCT circuit architecture and its simulation in VHDL. Results of the 2-D DCT simulation are included for two simple test patterns and verified by hand calculation, demonstrating the validity of the simulation. Shortcomings found in the simulation are described, together with suggestions for correcting them. In the future, the VHDL description of the 8 x 8 image block 2-D DCT can be further developed into structural and gate-level description, after which hardware circuit implementation can occur.			
20 DISTRIBUTION/AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED UNLIMITED <input type="checkbox"/> SAME AS REPORT <input type="checkbox"/> DTIC USERS		21 ABSTRACT SECURITY CLASSIFICATION Unclassified	
22a NAME OF RESPONSIBLE INDIVIDUAL Chin-Hwa Lee		22b TELEPHONE (Include Area code) (408) 646-2190	22c OFFICE SYMBOL EC

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted
All other editions are obsoleteSECURITY CLASSIFICATION OF THIS PAGE
Unclassified

Approved for public release; distribution is unlimited.

VHDL Behavioral Description
of Discrete Cosine Transform
in Image Compression

by

DENG, AN-TE
Lt. Col, Republic of China Army
B.S., Chung Cheng Institute of Technology, 1976

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN SYSTEM ENGINEERING

from the

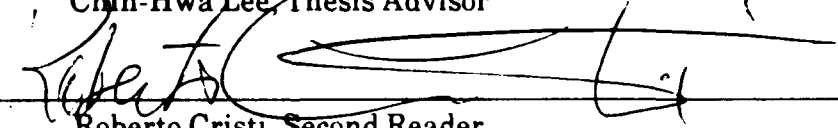
NAVAL POSTGRADUATE SCHOOL
September 1991

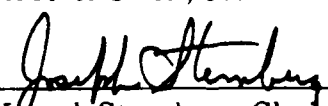
Author:


DENG AN-TE

Approved by:


Chin-Hwa Lee, Thesis Advisor


Roberto Cristi, Second Reader


Joseph Sternberg, Chairman
Department of Electronic Warfare

ABSTRACT

This thesis describes a VHSIC Hardware Description Language (VHDL) simulation of a hardware 8×8 Discrete Cosine Transform (DCT) which can be applied to image compression. A Top-Down Design approach is taken in the study, a discussion of DCT theory is presented, along with a description of the 1-D DCT circuit architecture and its simulation in VHDL. Results of the 2-D DCT simulation are included for two simple test patterns and verified by hand calculation, demonstrating the validity of the simulation. Shortcoming found in the simulation are described, together with suggestions for correcting them. In the future, the VHDL description of the 8×8 image block 2-D DCT can be further developed into structural and gate-level description, after which hardware circuit implement can occur.



Accession For	
NTIS SP4&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
Availability Codes	
A-1	

TABLE OF CONTENTS

I. INTRODUCTION	1
A. LITERATURE BACKGROUND	1
B. OBJECTIVE	1
C. RATIONALE FOR USING VHDL TO DESCRIBE THE CIRCUIT .	2
D. OVERVIEW OF THE THESIS	2
II. BASIC DISCRETE COSINE TRANSFORM THEORY	4
A. DISCRETE COSINE TRANSFORM IN IMAGE COMPRESSION . .	4
1. Rationale for using Discrete Cosine Transform	4
2. Formulae of the Discrete Cosine Transform	5
B. ALGORITHM FOR 8 BY 8 IMAGE DISCRETE COSINE TRANSFORM	8
1. Methodology of 2-D DCT	8
2. Principle of distributed arithmetic	10
3. Methodology for forming the ROM storage	11
4. Exploiting the symmetry in DCT to save storage in ROM	12
III. A STRUCTURAL ARCHITECTURE FOR THE 1-D DCT	16

A.	8 × 8 IMAGE BLOCK 1-D DCT CIRCUIT ARCHITECTURE . . .	16
B.	TRANSPOSE RAM ARCHITECTURE	20
IV.	VHDL BEHAVIORAL DESCRIPTION OF THE 1-D DCT COMPONENT .	22
A.	BLOCK DIAGRAM DESCRIPTION	22
B.	BI-TO-DI AND DI-TO-BI VHDL PACKAGE	23
C.	CLOCK GENERATOR MODULE (CLOCK_GE)	26
D.	PARALLEL SHIFT REGISTER MODEL (LOAD).	27
E.	SHIFT-TWO-REGISTER MODEL (SHIFT).	29
F.	2-BIT ADDER/SUBTRACTOR MODEL (ADDSUB)	31
G.	SHIFT REGISTER MODEL (REG)	33
H.	READ ONLY MEMORY MODEL (ROM)	34
I.	SHIFT RIGHT 1-BIT REGISTER MODEL (SHI_1)	35
J.	ADDER/SUBTRACTOR-G MODEL (ADD_G)	36
K.	SHIFT REGISTER-H MODEL (REG_H)	40
L.	16-BIT ADDER_I MODEL (ADD_I)	40
M.	SHIFT RIGHT 2-BIT REGISTER MODEL (SHI_2)	42
N.	PARALLEL LOAD SERIAL SHIFT REGISTER MODEL (RESET)	43
O.	TEST BENCH	45
V.	SIMULATION OUTPUT ANALYSIS AND EXPERIENCE	46

A.	FORMATION OF ROM STORAGE VALUES	46
B.	SIMULATION AND TESTING IMAGE PATTERN (I)	47
C.	SIMULATION AND TEST OF IMAGE PATTERN (II)	57
D.	RESULT ANALYSIS	60
E.	EXPERIENCE	63
1.	Input Data Sequential Order error	63
2.	Formation of 2-bit Adder in VHDL source code	64
3.	No Timing control in Add_i Model	64
4.	"Set" control in Test Bench	65
5.	Signals cannot be used as variables in VHDL	66
6.	Preventing Negative Zero occurrences in Pack1	66
VI.	CONCLUSION	67
	APPENDIX A. 12-BIT 1-D DCT VHDL SOURCE CODES	69
	APPENDIX B. 16-BIT 1-D DCT VHDL SOURCE CODE	105
	APPENDIX C. MATLAB PROGRAM OF DECIMAL-BINARY CONVERSION	114
	APPENDIX D. STRUCTURAL 1-D DCT HAND CALCULATION	115

APPENDIX E. FORMATION OF 2-BIT ADDER	121
A.TWO BIT ADDER TRUTH TABLE	121
LIST OF REFERENCES	125
INITIAL DISTRIBUTION LIST	126

LIST OF TABLES

Table I:	Multiplication Coefficients	46
Table II:	8 × 8 image pixel values of Pattern (I)	49
Table III:	1-D DCT spectral coefficients of Pattern (I) in VHDL simulation	50
Table IV:	1-D DCT coefficients of pattern (I) using Spider Subroutine . . .	50
Table V:	Transposed 1-D DCT coefficients of pattern (I) in VHDL simulation	51
Table VI:	2-D DCT spectral coefficients of pattern (I) in VHDL simulation	51
Table VII:	Table V in integer values	52
Table VIII:	2-D DCT spectral coefficients of pattern (I) using Spider Subroutine	52
Table IX:	2-D DCT coefficients of pattern (I) using direct calculation . . .	53
Table X:	16-bit binary number representation of table (V)	54
Table XI:	Serial 2-bit addition/subtraction output	55
Table XII:	2-D DCT coefficients of pattern (I) using manual calculation . .	56
Table XIII:	8 × 8 image block pixel values of pattern (II)	57
Table XIV:	1-D DCT coefficients of pattern (II) using VHDL simulation . .	58
Table XV:	2-D DCT coefficients of pattern (II) using VHDL simulation . .	58
Table XVI:	Pattern II 1-D DCT coefficients using Spider Subroutine	59

Table XVII: 2-D DCT coefficients of pattern (II) using floating point calculation	60
Table XVIII: Equivalent decimal numbers of table (VI)	61
Table XIX: Equivalent decimal numbers of table (XII)	61
Table XX: Truth table of 2-bit adder	121
Table XXI: (Table XX) continue	122

LIST OF FIGURES

Fig. 1	2-D DCT Block Diagram	9
Fig. 2	Architecture of 1-D DCT	16
Fig. 3	1-D DCT block diagram	22
Fig. 4	clock_ge block diagram	26
Fig. 5	Serial load parallel shift register block diagram	27
Fig. 6	Shift two register block diagram	29
Fig. 7	2-bit add/sub block diagram	31
Fig. 8	"adsu" flow chart	32
Fig. 9	shift register (reg) block diagram	33
Fig. 10	ROM block diagram	35
Fig. 11	Shi_1 register block diagram	36
Fig. 12	Add_g block diagram	37
Fig. 13	Shift register_g block diagram	40
Fig. 14	16-bit add_i block diagram	41
Fig. 15	Shift right 2-bit register block diagram	42
Fig. 16	Parallel shift serial output register block diagram	44
Fig. 17	Block diagram of Test Bench	45
Fig. 18	Pattern (I) 8×8 image block	48
Fig. 19	U0 hand calculation	115
Fig. 20	V1 hand calculation	116

Fig. 21	V3 hand calculation	117
Fig. 22	U4 hand calculation	118
Fig. 23	V5 hand calculation	119
Fig. 24	V7 hand calculation	120
Fig. 25	Karnaugh map reduction	123

ACKNOWLEDGEMENTS

Many of the ideas in this thesis are based on the experience of my advisor, Dr. Chin-Hwa Lee, who has labored with me through the chapters. Many thanks go to Dr. Lee for his patience and valuable advises. Also I am very grateful to Dr. Roberto Cristi for his comments on my thesis.

It has been a pleasure sharing with Dr. Pat Pauley, who not only has been a supporting force, but also has proofread my thesis in no time during her busiest hours.

I owe special thanks to those who have suffered with me through the writing process -- my family: my wife Vicky, my daughter Bobo and my son Joshua.

Do you not know? Have you not heard? The Lord is the everlasting God, the Creator of the ends of the earth. He will not grow tired or weary, and his understanding no one can fathom. He gives strength to the weary. Even youths grow tired and weary; and young men stumble and fall; but those who hope in the Lord will renew their strength. They will soar on wings like eagles; they will run and not grow weary, they will walk and not be faint.

Isaiah 40:28-31

I. INTRODUCTION

A. LITERATURE BACKGROUND

This thesis is basically developed from the paper "An 8×8 Discrete Cosine Transform Chip with Pixel Rate Clock" by D'Luna, L. J. [Ref. 1]. The original paper introduced the algorithm and implementation of one-dimensional (1-D) as well as two-dimensional (2-D) Discrete Cosine Transform (DCT) where the principle of distributed arithmetic is used. According to the algorithm introduced, hardware circuit architecture was implemented.

Another very important aspect discussed in this thesis is the implementation of a "Top-Down Design" concept that uses Very High Speed Integrate Circuit (VHSIC) Hardware Description Language [Ref. 4-8] as a tool. "Top-Down Design" is a kind of design that describes the given algorithm with a high level language first. After the algorithm is described, the structural architecture is described next. Finally this structural description is developed into hardware circuit. VHDL facilitates the algorithm description, structural description as well as hardware circuit simulation.

B. OBJECTIVE

The purpose of this thesis is to describe the behavior of the implemented architecture of the algorithm mentioned above with VHSIC Hardware Description Language (VHDL). It was simulated on a workstation in order to analyze the

characteristics. In the process of describing the behavior of this structural architecture, complicated hardware circuits are developed in behavior models. This is usually the first step in a "Top-Down Design" task. The objective is to use a DCT implementation as an example to study the "Top-Down Design" methodology.

C. RATIONALE FOR USING VHDL TO DESCRIBE THE CIRCUIT

In the past, VHSIC design was dominated by bottom-up design methodologies where hardware circuit details were established and produced before the system was constructed [Ref. 4]. This methodology is very useful in dealing with small circuits. However, when the system gets complicated, bottom-up design methodology is more difficult to handle. In this work, a high-level, top-down design approach is taken. Initially, a description of the algorithm is written. Later on, a detailed architecture is described. All are done in VHDL. VHDL is a hierarchical hardware description language which supports mixed-level simulation. This thesis shows the beginning steps for a "Top-Down Design" approach. The 8×8 image block DCT algorithm were implemented into a behavior model and a structural model. VHDL were used here to accomplish the initial design of the 1-D Discrete Cosine Transform implementation.

D. OVERVIEW OF THE THESIS

There are six chapters in this thesis. The first chapter is an introduction to the literature background, the objective, and the reasons for using the VHDL. Chapter II introduces the algorithm of Discrete Cosine Transform and the principle of distributed arithmetic. Chapter III examines the components of the structural architecture. Chapter

IV gives the actual VHDL behavioral description of the components, its actual circuit block diagram, and its connections. Chapter V analyzes the simulation results and gives some experience on design problems. The last chapter is the conclusion.

II. BASIC DISCRETE COSINE TRANSFORM THEORY

A. DISCRETE COSINE TRANSFORM IN IMAGE COMPRESSION

1. Rationale for using Discrete Cosine Transform

Image transmission or storage usually deals with a large amount of digital data. There are usually 512×512 pixels in a monochrome picture. If one pixel needs 8 bits to represent its information, transmitting a monochrome picture means that more than two megabits ($512 \times 512 \times 8 = 2,097,152$) of digit data need to be transmitted. There are many ways to do coding, compressing huge amounts of data to reduce the transmission bandwidth and the amount of storage space required. Among these methods, transform domain compression is an effective way to eliminate the redundant information in images, since image data are usually highly correlated.

Image transformation is used to extract a small number of significant coefficient values from the original image, by mapping the image data onto a two-dimensional spectrum. Each coefficient in the transform domain represents some amount of energy of the spectral component. The original spatial image can then be recovered back from these coefficients, since each image has its own specific spectral pattern. After the transformation, there are only a few coded values required to describe the original image. Consequently, it is possible to save bits during transmission and storage.

The Fourier transform algorithm has been applied to image processing for a long time, since it possesses many desirable analytic properties. But, it has two major

drawbacks. First, the computation of the Fourier transform involved complex numbers rather than real numbers. Secondly, the decreasing rate of spectrum energy as frequency increases is low. This low decreasing rate in the spectrum is a very significant disadvantage in image coding.

The Discrete Cosine Transform (DCT) has the advantage of involving only real number computations. It is well suited for image data compression. Consequently, 8×8 image blocks of two dimensional cosine transforms have been adopted as an international standard draft (JPEG) [Ref. 1]. This thesis concentrates on studying the Discrete Cosine Transform and building a circuit for 8×8 image blocks.

2. Formulae of the Discrete Cosine Transform

The general formula of a one-dimensional Discrete Fourier Transform (1-D DCT) is expressed as

$$Z_k = \sum_{i=0}^{N-1} X_i C_{ik} \quad (1)$$

where Z_k is the transform of X_i , C_{ik} is the *forward transformation kernel*, and i and k range from 0 to $N - 1$. The inverse transform of the 1-D DCT is given by the relation

$$X_i = \sum_{k=0}^{N-1} Z_k h_{ik} \quad (2)$$

where h_{ik} is the *inverse transformation kernel*. The characteristic of the transform is determined by its transformation kernel properties.

The 1-D DCT forward kernel is given by

$$C_{i0} = \frac{1}{\sqrt{N}} \quad (3)$$

$$C_{ik} = \sqrt{\frac{2}{N}} \cos \frac{(2i + 1)k\pi}{2N} \quad (4)$$

Substituting Eq. (3) and (4) into Eq. (1) yields

$$Z_0 = \frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} X_i \quad (5)$$

$$Z_k = \sqrt{\frac{2}{N}} \sum_{i=0}^{N-1} X_i \cos \frac{(2i + 1)k\pi}{2N} \quad (6)$$

where Z_k , $k = 0, 1, 2, \dots, N - 1$, is the 1-D DCT of $X(i)$.

The inverse kernel is of the same form as Eq. (3) and (4), so that the inverse DCT is expressed by the equation

$$X_i = \frac{1}{\sqrt{N}} Z_0 + \sqrt{\frac{2}{N}} \sum_{k=1}^{N-1} Z_k \cos \frac{(2i + 1)k\pi}{2N} \quad (7)$$

where $i = 0, 1, 2, \dots, N - 1$.

The two-dimensional forward DCT kernel is given as

$$C_{y00} = \frac{1}{N} \quad (8)$$

$$C_{ijkl} = \frac{2}{N} \left[\cos \frac{(2i+1)k\pi}{2N} \right] \left[\cos \frac{(2j+1)l\pi}{2N} \right] \quad (9)$$

where $i, j = 0, 1, \dots, N-1$, and $k, l = 1, 2, \dots, N-1$. The inverse kernel is also of this form. Thus, the two-dimensional DCT pair is expressed by

$$Z_{00} = \frac{1}{N} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} X_{ij} \quad (10)$$

$$Z_{kl} = \frac{2}{N} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} X_{ij} \left[\cos \frac{(2i+1)k\pi}{2N} \right] \left[\cos \frac{(2j+1)l\pi}{2N} \right] \quad (11)$$

where $k, l = 1, 2, \dots, N-1$, and

$$X_{ij} = \frac{1}{N} Z_{00} + \frac{2}{N} \sum_{k=1}^{N-1} \sum_{l=1}^{N-1} Z_{kl} \left[\cos \frac{(2i+1)k\pi}{2N} \right] \left[\cos \frac{(2j+1)l\pi}{2N} \right] \quad (12)$$

where $i, j = 0, 1, \dots, N-1$.

It can be seen that DCT transformation kernels are separable from Eqs. (3), (4), (8), and (9). Therefore, the two-dimensional forward or inverse transformation can be computed by applying two one-dimensional DCT operations successively.

B. ALGORITHM FOR 8 BY 8 IMAGE DISCRETE COSINE TRANSFORM

1. Methodology of 2-D DCT

Let x_{ij} denote an image pixel value, which is an n -bit number. The indices i and j represent the row and column location of the pixel, respectively. The $N \times N$ two-dimensional DCT can be expressed by

$$Z_{00} = \frac{1}{N} \sum_{j=0}^{N-1} \sum_{i=0}^{N-1} x_{ij} \quad (13)$$

$$Z_{kl} = \frac{2}{N} \sum_{j=0}^{N-1} \sum_{i=0}^{N-1} x_{ij} \cos \frac{(2i+1)k\pi}{2N} \cos \frac{(2j+1)l\pi}{2N} \quad k, l = 1, 2, \dots, N-1. \quad (14)$$

Z_{kl} is the spectral coefficient corresponding to the k^{th} horizontal frequency and l^{th} vertical frequency. In matrix notation, the inner summation is equivalent to a 1-D DCT computation on the columns of X . The outer summation is equivalent to a 1-D DCT computation on the rows of the inner summation results. C can be used to represent the 2-D DCT matrix. It has the 1-D DCT basis vectors which are elements C_{mk} (1-D DCT kernels), where

$$C_{m0} = \frac{1}{\sqrt{N}} \quad m = 0, 1, 2, \dots, N-1 \quad (15)$$

$$C_{mk} = \sqrt{\frac{2}{N}} \cos \frac{(2m+1)k\pi}{2N} \quad (16)$$

$m = 0, 1, 2, \dots, N-1; k = 1, 2, \dots, N-1$. Because the kernels of the DCT transformation can be separated, the 2-D matrix Z of 2-D DCT coefficients can be represented as

$$Z = [X'C]'C = C'XC. \quad (17)$$

This process can be realized in an architecture shown in Fig. 1 (referred to Ref. 1]).

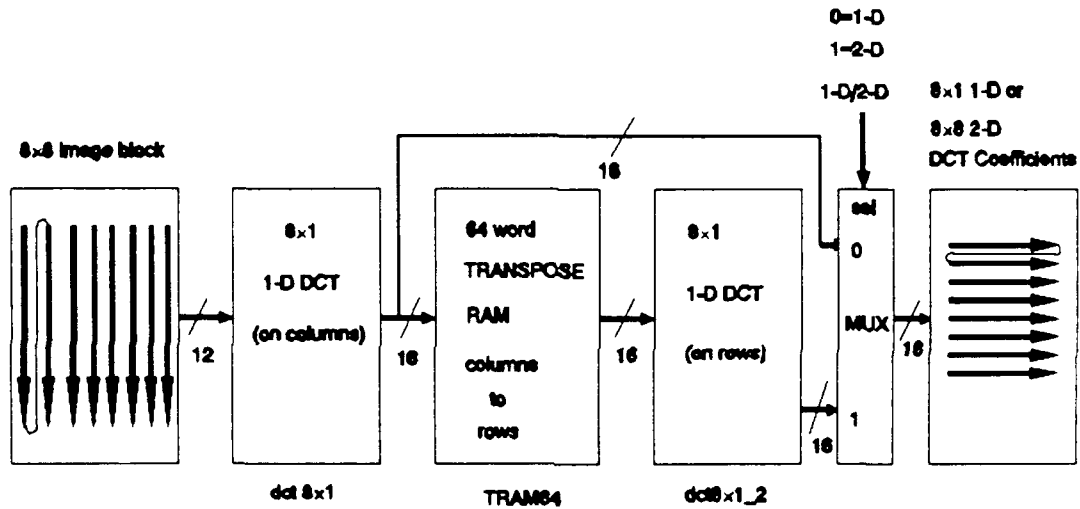


Fig. 1 2-D DCT Block Diagram

The $N \times N$ block of image X is input column by column first, and the 1-D DCT computation is done. This computation is carried out as shown in the square bracket of Eq.(17) for the j^{th} column (for $j = 0, 1, \dots, N - 1$). The result of this $N \times N$ matrix is then transposed for the second row by row 1-D DCT computation. This transpose is

done as described by term on the outside of the square brackets in Eq.(17). After the transposition, the same 1-D DCT computation involving the same transform matrix C is carried out again. The transpose step takes care of the column to row change operations of the data. The key operations involved here are the matrix transpose and the 1-D DCT computation.

2. Principle of distributed arithmetic

The implementation of the 1-D DCT studied here is based on the principle of distributed arithmetic. Using this principle, it is possible to implement the "bit calculation" into the chip design. "Bit multiplication" is simply carried out by using the input data bit pattern to address a Read Only Memory and by summing up all the results to obtain the "transposed spectral values". If Y_i ($Y_i = (y_{im})_{m=0}^{N-1}$) is the image pixel value represented by a row vector, then its 1-D DCT is

$$Z_{ik} = \sum_{m=0}^{N-1} y_{im} C_{mk} \quad i, k = 0, 1, \dots, N - 1. \quad (18)$$

Now the input data y_{im} can be represented in 2's complement notation with p-bit as

$$y_{im} = -y_{im}^{(p-1)} 2^{p-1} + \sum_{q=0}^{p-2} y_{im}^{(q)} 2^q \quad (19)$$

where $y_{im}^{(q)}$ is the q^{th} bit of the incoming image pixel values y_{im} which have a value of either 0 or 1. 2^q is the binary weight of the q^{th} bit. For example, if the input data is a 2's

complement 8-bit pattern then $y_{im} = -y_{im}^{(7)} \times 2^7 + y_{im}^{(0)} \times 2^0 + y_{im}^{(1)} \times 2^1 + y_{im}^{(2)} \times 2^2 + y_{im}^{(3)} \times 2^3 + y_{im}^{(4)} \times 2^4 + y_{im}^{(5)} \times 2^5 + y_{im}^{(6)} \times 2^6$. Substituting Eq. (19) into Eq. (18)

$$Z_{ik} = -\sum_{m=0}^{N-1} c_{mk} y_{im}^{(p-1)} 2^{p-1} + \sum_{q=0}^{p-2} \sum_{m=0}^{N-1} c_{mk} y_{im}^{(q)} 2^q \quad (20)$$

$$Z_{ik} = -F_{ik}(C_k, Y_i^{(p-1)}) 2^{p-1} + \sum_{q=0}^{p-2} F_{ik}(C_k, Y_i^{(q)}) 2^q \quad (21)$$

where F_{ik} is a function of the vectors C_k and $Y_i^{(q)}$ and is represented as

$$F_{ik}(C_k, Y_i^{(q)}) = \sum_{m=0}^{N-1} c_{mk} y_{im}^{(q)} \quad \text{for } q = 0, 1, 2, \dots, p-1. \quad (22)$$

Its binomial form can be shown as

$$F_{ik}(C_k, Y_i^{(q)}) = c_{0k} y_{i0}^{(q)} + c_{1k} y_{i1}^{(q)} + \dots + c_{N-1,k} y_{i,N-1}^{(q)} \quad (23)$$

where, $q = 0, 1, \dots, p-1$.

3. Methodology for forming the ROM storage

In Eq.(23), c_{mk} are 1-D DCT basis (kernels) vectors used as multiplication coefficients. They are converted from decimal numbers to the 2's complement notation used in this thesis. $y_{im}^{(q)}$ are the bit patterns represented in 2's complement form of the N data points y_{im} . Because the basis vectors are fixed value coefficients and F_{ik} are functions of the basis vectors and the binary bit patterns, the values of F_{ik} (with a fixed

k) for all possible N bit patterns ($y_m^{(q)}$, $m = 0, 1, 2, \dots, N-1$) can be calculated and stored in Read Only Memory (ROM) according to Eq.(22) and Eq.(23). The N-bit pattern changes with time according to the incoming data $y_m^{(q)}$ ($m = 0, 1, 2, \dots, N-1$). This bit pattern will form an address to access the ROM to extract the corresponding $F_k(C_k, Y_i^{(q)})$ value.

From Eq.(20) and Eq.(21), the corresponding 1-D DCT spectral coefficient Z_k can be computed by shifting and adding the F_k values stored in the ROM. In Eq. (21), F_k is a function of the corresponding basis column vector C_k for $k = 0, 1, 2, \dots, N-1$. F_k is different from each other as k varies. The incoming data vector Y_i is the same for the multiplication coefficients involved for all values of k . It is possible to build up N separate memory banks of multiplication coefficients and compute the N 1-D DCT spectral coefficients Z_k ($k = 0, 1, 2, \dots, N-1$) in parallel or concurrently.

4. Exploiting the symmetry in DCT to save storage in ROM

Here, 8×8 image blocks are used, so $N = 8$. The incoming data has 8 bits. This means $2^8 = 256$ possible bit patterns will be formed into addresses. There shall be 256 corresponding multiplication coefficient sum stored in the ROM for each of the 8 DCT spectral coefficients. However, advantage can be taken of the symmetry in the DCT basis vectors. It can be shown that

$$C_{mk} = C_{N-1-m,k} \quad \text{for } k = 0, 2, \dots, N-2 \quad (k \text{ even}). \quad (24)$$

For example,

$$C_{02} = \sqrt{\frac{2}{8}} \cos \frac{2\pi}{16} = \sqrt{\frac{2}{8}} \cos \frac{30\pi}{16} = C_{72} \quad (25)$$

where c_{mk} is defined by Eq. (15) and Eq. (16). And the following can be proven,

$$C_{mk} = -C_{N-1-m,k} \quad \text{for } k = 1, 3, \dots, N-1 \text{ (} k \text{ odd)} \quad (26)$$

For example,

$$C_{01} = \sqrt{\frac{2}{8}} \cos \frac{\pi}{16} = -\sqrt{\frac{2}{8}} \cos \frac{15\pi}{16} = -C_{71}. \quad (27)$$

Hence, Eq. (18) can be reduced to

$$Z_{ik} = \sum_{m=0}^{N/2-1} (y_{im} + y_{i,N-1-m}) C_{mk}$$

$$\text{where } k = 0, 2, \dots, N-2 \text{ (} k \text{ even)} \quad (28)$$

and,

$$Z_{ik} = \sum_{m=0}^{N/2-1} (y_{im} - y_{i,N-1-m}) C_{mk}$$

$$\text{where } k = 1, 3, \dots, N-1 \text{ (} k \text{ odd).} \quad (29)$$

Equations (22) and (23) then can be reduced to

$$F_{ik}(C_k, Y_i^{(q)}) = \sum_{m=0}^{N/2-1} C_{mk} (y_{im} + y_{i,N-1-m})^{(q)}$$

$$\text{where } k = 0, 2, 4, \dots, N-2 \quad (30)$$

$$F_{ik}(C_k, Y_i^{(q)}) = \sum_{m=0}^{N/2-1} C_{mk} (Y_{im} - Y_{i,N-1-m})^{(q)}$$

$$\text{where } k = 1, 3, 5, \dots, N-1. \quad (31)$$

From the above equations, it is possible to add or subtract the incoming data points before memory access and reduce the number of distinct data values in ROM from N to $N/2$. The total number of bit patterns is now only $2^{N/2} = 2^4 = 16$. Only a 16 word ROM is necessary for each of the 8 DCT coefficients, and therefore a total of $16 \times 8 = 128$ word ROM is required. This savings of ROM storage is significant compared to the cost of using adders and subtractors in a different architecture. Since there is only one particular bit pattern (those bits which have the same binary weight) at a time allowed to address the ROM, and bit pattern changes according to the serially coming

data, the addition and subtraction can be done in a bit serial fashion. This advantage is exploited in the chip implementation discussed in the next chapter.

III. A STRUCTURAL ARCHITECTURE FOR THE 1-D DCT

A. 8×8 IMAGE BLOCK 1-D DCT CIRCUIT ARCHITECTURE

The 1-D DCT architecture studied previously is shown in Fig. 2 [Ref. 1]. There

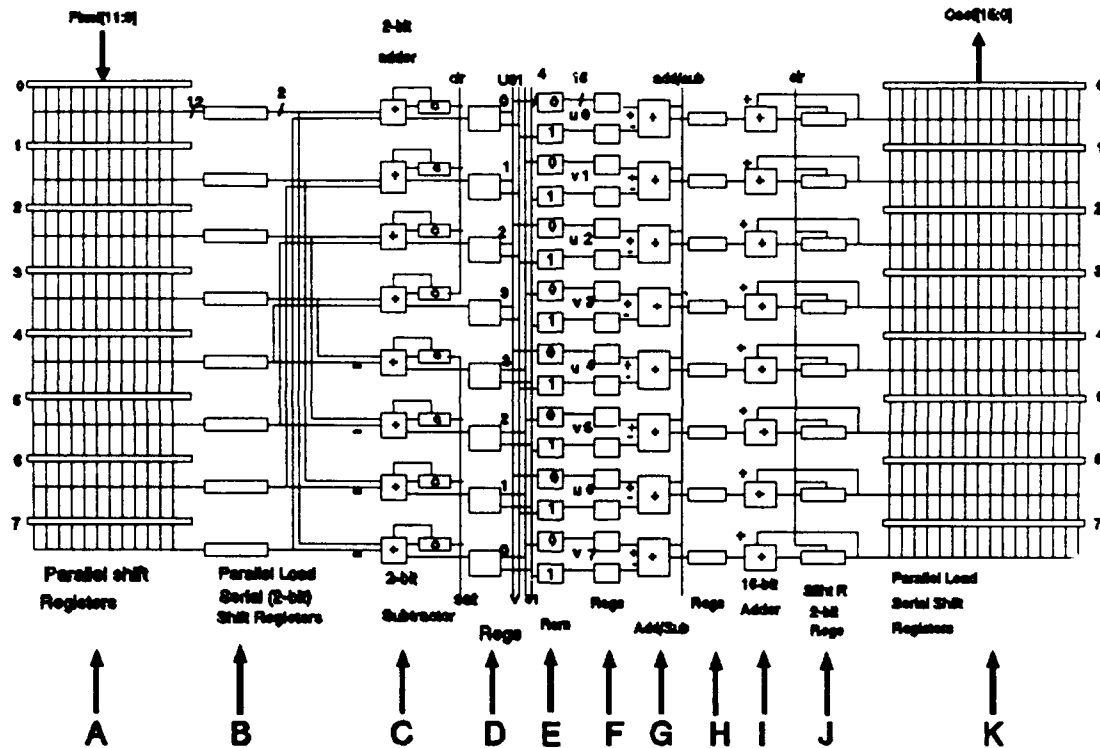


Fig. 2 Architecture of 1-D DCT

are 8 slices parallel to each other corresponding to the 8 DCT coefficients which are computed concurrently. First, 12-bit pixels $AI(11:0)$ are put column by column into the "serial-in-parallel-out" shift register (A). This sequence needs 8 clock cycles to complete.

After the 8th clock, the shift registers output the data into the "parallel load 2-bit serial shift register" (B) at once. This is completed at the 9th clock cycle. At the same time, the serial-in-parallel-out shift registers also get their new incoming data. The data stored in the B shift register has to be added or subtracted according to Eqs. (30) and (31) in order to reduce the ROM storage. In order to make Eqs. (30) and (31) more understandable, they are expanded as below

$$F_{ik}(C_k, Y_i^{(q)}) = \sum_{m=0}^{N/2-1} C_{mk}(Y_{im} + Y_{i,N-1-m})^{(q)}$$

$$\begin{aligned} & \begin{matrix} m = 0 & m = 1 & m = 2 & m = 3 \end{matrix} \\ & = C_{00}(Y_{i0} + Y_{i7})^{(q)} + C_{10}(Y_{i1} + Y_{i6})^{(q)} + C_{20}(Y_{i2} + Y_{i5})^{(q)} + C_{30}(Y_{i3} + Y_{i4})^{(q)} \text{----} k = 0 \\ & + C_{02}(Y_{i0} + Y_{i7})^{(q)} + C_{12}(Y_{i1} + Y_{i6})^{(q)} + C_{22}(Y_{i2} + Y_{i5})^{(q)} + C_{32}(Y_{i3} + Y_{i4})^{(q)} \text{----} k = 2 \\ & + C_{04}(Y_{i0} + Y_{i7})^{(q)} + C_{14}(Y_{i1} + Y_{i6})^{(q)} + C_{24}(Y_{i2} + Y_{i5})^{(q)} + C_{34}(Y_{i3} + Y_{i4})^{(q)} \text{----} k = 4 \\ & + C_{06}(Y_{i0} + Y_{i7})^{(q)} + C_{16}(Y_{i1} + Y_{i6})^{(q)} + C_{26}(Y_{i2} + Y_{i5})^{(q)} + C_{36}(Y_{i3} + Y_{i4})^{(q)} \text{----} k = 6 \end{aligned}$$

$$F_{ik}(C_k, Y_i^{(q)}) = \sum_{m=0}^{N/2-1} C_{mk}(Y_{im} - Y_{i,N-1-m})^{(q)}$$

$$\begin{aligned} & \begin{matrix} m = 0 & m = 1 & m = 2 & m = 3 \end{matrix} \\ & = C_{01}(Y_{i0} - Y_{i7})^{(q)} + C_{11}(Y_{i1} - Y_{i6})^{(q)} + C_{21}(Y_{i2} - Y_{i5})^{(q)} + C_{31}(Y_{i3} - Y_{i4})^{(q)} \text{----} k = 1 \\ & = C_{03}(Y_{i0} - Y_{i7})^{(q)} + C_{13}(Y_{i1} - Y_{i6})^{(q)} + C_{23}(Y_{i2} - Y_{i5})^{(q)} + C_{33}(Y_{i3} - Y_{i4})^{(q)} \text{----} k = 3 \\ & = C_{05}(Y_{i0} - Y_{i7})^{(q)} + C_{15}(Y_{i1} - Y_{i6})^{(q)} + C_{25}(Y_{i2} - Y_{i5})^{(q)} + C_{35}(Y_{i3} - Y_{i4})^{(q)} \text{----} k = 5 \\ & = C_{07}(Y_{i0} - Y_{i7})^{(q)} + C_{17}(Y_{i1} - Y_{i6})^{(q)} + C_{27}(Y_{i2} - Y_{i5})^{(q)} + C_{37}(Y_{i3} - Y_{i4})^{(q)} \text{----} k = 7 \end{aligned}$$

The numbers above the expanded equation represent the index m , and the numbers on the right side are the index k . C_{mk} are multiplication coefficients. The bit addition/subtraction is determined according to whether k is an even or odd number.

Registers B must be emptied in less than 8 clock cycles in order to receive new data coming from registers A. Each datum is 12 bits in length. If a single bit is coming out of registers B, it will take 12 clock cycles to empty the register. This will cause collision during the addition and subtraction of the data. There are two ways to solve this problem ; either to clock register B twice as fast or to shift out data 2 bits at a time. The latter alternative has been chosen for the reasons of convenient design and easy system considerations. The shifted 2-bit data is added or subtracted in the "2-bit adder/subtractor" C. Their output is stored in the shift registers D which split the least significant bit and most significant bit (binary weight $q = 0$ and $q = 1$) into two output lines.

Next comes the question as to where the output data of the adders and subtractors should go to address the ROM. How should the values in the ROM be arranged? It is shown in the above expanded equations that all the adder outputs which is designated as $(U_0(0:3)$ and $U_1(0:3)$ (Refer to Fig. 2). They are the 4 bits patterns which are the sum of the two adjacent bit $Y_{im}^{(q)}$. $q = 0$ represents LSB bit and $q = 1$ represents MSB bit in Eqs (20) and (21). $(U_0(0:3)$ and $U_1(0:3)$ should be multiplied by the coefficients C_{mk} , where $k = 0, 2, 4, 6$. All the two adjacent difference output $V_0(3:0)$ and $V_1(3:0)$ should be multiplied by the coefficients C_{mk} , where $k = 1, 3, 5, 7$. As a result, the four adders and subtractors output bit patterns form a 4-bit address to access the corresponding

accumulated sum of the coefficients C_{mk} , $k = 0, 1, \dots, 7$ which are stored in ROM E. This step will accomplish the 1-D DCT coefficient multiplication. The output of the ROM is first latched in register F, and then adder/subtractor G will calculate the sum of the "2-bit" spectral coefficient values according to Eq (21). The LSB ($q = 0$) values are shifted to the right one position and added to the $q = 1$ values. This addition will continue until the last bit pattern (12^{th}) of the incoming column data. According to Eq. (19), the incoming data have been represented in 2's complement notation, so the most significant bit's value should be subtracted from all the previous summations. This is done by changing the add/sub control line of G into subtraction at the clock cycle of the last bit pattern for each column of data.

The 2-bit sum or difference results of G are stored into register H and then sent to the accumulator I and J. The accumulator consists of one "16-bit adder" and a "shift right 2-bit register". The value stored in ROM E is a 16-bit word. The 16-bit adder I adds the previous 2-bit right shifted value (output of J) to the incoming value (output of H). The resulting value then is output to J register to do the 2-bit right shift. This process will accomplish the computation of Eq. (21) as index q varies from 0 to $p-1$ in 2 bit increments. One thing has to be noted with caution; the initial value in the shift right 2-bit registers for every incoming column of data should be zero. Otherwise, the previous column values would accumulate. To avoid this, just clear the shift right 2-bit register at the beginning of the accumulation of every column group.

After 8 clock cycles, the accumulated values are parallel loaded into register K. Similar to register A but in the reverse direction, register K puts out the 1-D DCT

spectral coefficients column by column. These 1-D DCT coefficients are then transposed by the transpose RAM (TRAM) according to Eq.(17). The transpose RAM is described in the next section. After the transpose RAM, 1-D DCT coefficients are then input into again the same 1-D DCT architecture. The only difference now is that the registers A and B have to be expanded from 12 bits to 16 bits for the second transform.

B. TRANSPOSE RAM ARCHITECTURE

According to Eq. (17), the purpose of the "transpose RAM" is to change the 8×8 1-D DCT coefficient block's columns into rows; and rows into columns. The coefficient values are generated from the 1-D DCT architecture column by column. First, these values are put into a RAM while the transposed values are written. Therefore, the transpose RAM must have the capability of reading in the 1-D DCT values and writing out the transposed values in the same cycle. How can this be done?

The coefficient values come out of the 1-D DCT architecture in serial order; the 0, 1, 2,..., 7 coefficients of the first column of the 8×8 block come in first and then the 0,1,... 7 coefficients of the second column and the third column and so on. This order is a long stream of coefficients 0,1,... 63 for each 8×8 image block. After storing them in the RAM, the coefficients must be read out in groups of 8 values in the order of 0, 8, 16,..., 56; 1, 9, 17,..., 57; 2, 10, 18,..., 58; 3, 11, 19,..., 59; 4, 12, 20,..., 60; 5, 13, 21,..., 61; 6, 14, 22,..., 62; 7, 15, 23,..., 63 to achieve the transpose operation. In the same cycle, just after reading out the first block of transposed values, the coefficient values of the second block can be written into those locations. It is just

like reading block 1_0 (first 8×8 block position 0) and writing block 2_0 (second 8×8 block position 0), reading block 1_8 and writing block 2_1, reading block 1_16 and writing block 2_2, and so on. In order to achieve the transpose of the second block, the sequence for reading out block 2 must be in the order of 0, 1, 2,... 63. When reading out the coefficients of block 2, the third block coefficients are being written into the same locations just after read out. The order is just like reading block 2_0 and writing block 3_0, reading block 2_1 and writing block 3_1, reading block 2_2 and writing block 3_2, and so on. Notice the sequential order is 0, 1, 2,...63 first, and then 0, 8, 16,..., and then again in the sequential order of 0, 1, 2,...63, and so on.

As shown before the structural architecture design is based on the principle of distributed arithmetic, and it is data-path oriented. The methodology to describe this architecture in VHDL and to simulate it on a computer are discussed in the next chapter.

IV. VHDL BEHAVIORAL DESCRIPTION OF THE 1-D DCT COMPONENT

A. BLOCK DIAGRAM DESCRIPTION

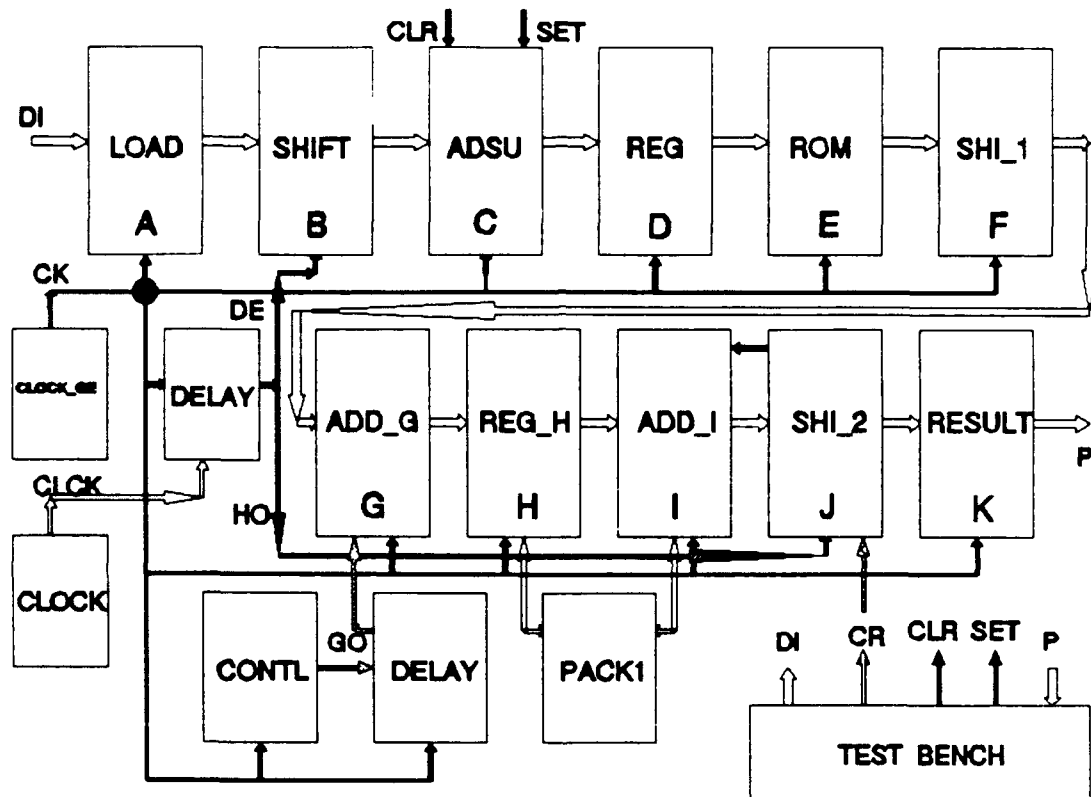


Fig. 3 1-D DCT block diagram

The block diagram of the 1-D DCT shown in Fig. 3 can be described in models using VHDL. The block diagram shown here includes a 1-D DCT system discussed in chapter III and the additional clock generators, delay lines, control line, package 1, and test bench. There are minor differences between this diagram and the architecture described in the previous chapter. What is taken into consideration when simulating this

system in VHDL is that a signal flow latency will occur. Therefore, a delay line is necessary to change the clock triggering time and solve this latency problem. Additionally, the architecture in the previous chapter does not make it clear when to control the add/sub register G and fulfill the calculation of summing 2's complement values. It is shown here that the control line generating this control bit is triggered by the delayed clock.

From the modeling point of view, it is rather complicated to build up a 16-bit adder in VHDL following the usual arithmetic logics. The easiest approach is to convert the 16-bit binary coefficient values into integer numbers and then do the addition or subtraction in integers. After the integer addition or subtraction, the integers are simply converted back to binary values. This conversion task is accomplished by functions in package 1. A VHDL package is a collection of functions and procedures. Of course, some overflow/underflow situations are expected to occur during these conversions. One last thing to note in Figure 3 is that the test bench module controls all the signal flow, the input data, and the output data, and it also simulates the whole design.

B. BI-TO-DI AND DI-TO-BI VHDL PACKAGE

the package 1 in VHDL is shown below,

```
package pack1 is -- Package declaration
procedure bi_to_in -- Procedure 1 changes 16 bits binary into integer
    (variable x : bit_vector(15 downto 0);
     variable y : out integer);
procedure in_to_bi --Procedure 2 changes integer into binary
    (variable m : in integer;
     variable n : out bit_vector(15 downto 0));end pack1;
package body pack1 is -- Package body declaration
```

```

procedure bi_to_in  -- First procedure that changes bits to integer
  (variable x : bit_vector(15 downto 0);
   variable y : out integer) is
   variable sum : integer := 0;
   variable p : bit_vector(15 downto 0);
begin
  p := x;
  if p(15) = '1' then          -- Change negative value to positive
    for i in 0 to 14 loop
      if p(i) = '1' then
        for i in 0 to 13 loop
          p(i+1) := not p(i+1);
        end loop; exit;
      end if;
    end loop;
    for k in 0 to 14 loop      -- Integer conversion
      if p(k) = '1' then
        sum := sum + 2**k;
      end if;
    end loop;
    y := -sum;                  -- Convert back to negative value
  else
    for l in 0 to 14 loop      -- Positive value conversion
      if p(l) = '1' then
        sum := sum + 2**l;
      end if;
    end loop;
    y := sum;
  end if;
end bi_to_in; -- end of procedure 1

```

```

procedure in_to_bi -- Second procedure that changes integer to bits
  (variable m : in integer;
   variable n : out bit_vector(15 downto 0)) is
   variable temp_a : integer := 0;
   variable temp_b : integer := 0;
   variable w : bit_vector(15 downto 0);
begin
  if m < 0 then
    temp_a := -m; -- Take the absolute value of negative values
  else
    temp_a := m;
  end if;

```

```

for i in 14 downto 0 loop      -- Binary conversion
    temp_b := temp_a/(2**i);
    temp_a := temp_a rem (2**i);
    if (temp_b = 1) then
        w(i) := '1';
    else
        w(i) := '0';
    end if;
end loop;
if m > 0 then
    w(15) := '0';              -- Assign positive sign bit
else
    w(15) := '1';              -- Assign negative sign bit
    for k in 0 to 14 loop
        if w(k) = '1' then
            for k in 0 to 13 loop -- Invert negative bits to 2's complement
                w(k+1) := not w(k+1);
            end loop; exit;
        end if;
    end loop;
end if;
if w(14)='0' and w(13)='0' and w(12)='0' and w(11)='0'
and w(10)='0' and w(9)='0' and w(8)='0' and w(7)='0'
and w(6)='0' and w(5)='0' and w(4)='0' and w(3)='0'
and w(2)='0' and w(1)='0' and w(0)='0'
then
    w(15) := '0';              -- Avoid negative zero
end if;
-----
n := w;
end in_to_bi; -- end of procedure 2
end pack1; -- end of procedure

```

This VHDL package used in the simulation is basically similar to any other high-level language subroutine involving specific shared operations. The difference here is that it is possible to gather several different procedures or functions together in one package. The pack1 here consists of two procedures -- bi_to_in and in_to_bi. Bi_to_in converts the 16-bit binary numbers (represented in 2's complement notation) into positive

or negative integers. The `in_to_bi` procedure converts the positive or negative integers back to 2's complement 16-bit binary numbers. Note that in the 2's complement number system used here, there are only 16 bits including one sign bit. In overflow situations, the digits that overflow will be truncated.

C. CLOCK GENERATOR MODULE (CLOCK_GE)

The block diagram of the "clock_ge" is shown in Figure 4.

The interface connection (port map in VHDL) has also been shown. This tells how the circuit can be connected to the test bench. The VHDL source code of the `clk.vhd` is shown below,

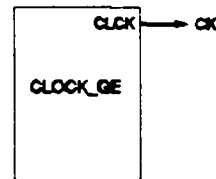


Fig. 4 clock_ge block diagram

```
entity clock_ge is -- Entity
    -- declaration
    port(CLCK :inout bit);
end clock_ge;
architecture clk_ctl of clock_ge is -- Architecture declaration
begin
    process(CLCK) -- Process declaration
        variable I : integer := 0;
    begin -- Process begin
        CLCK <= not CLCK after 5 ns; -- Switching clock generation
        I := I + 1;
        assert I <= 80 -- Assertion terminates the infinite process
            report "job done"
            severity Error;
    end process; -- End of process
end clk_ctl; -- End of architecture
```

There is a sensitivity signal "CLCK" in the source code which provides the clock for all the circuits. The initial value of CLCK is "0." Its value is changed into "1" after

5 ns. Since a process in VHDL basically is an infinite loop, it is necessary to use an "assert" instruction to terminate the process. By changing a counter value "I", the job can be terminated appropriately after 80 iterations.

D. PARALLEL SHIFT REGISTER MODEL (LOAD).

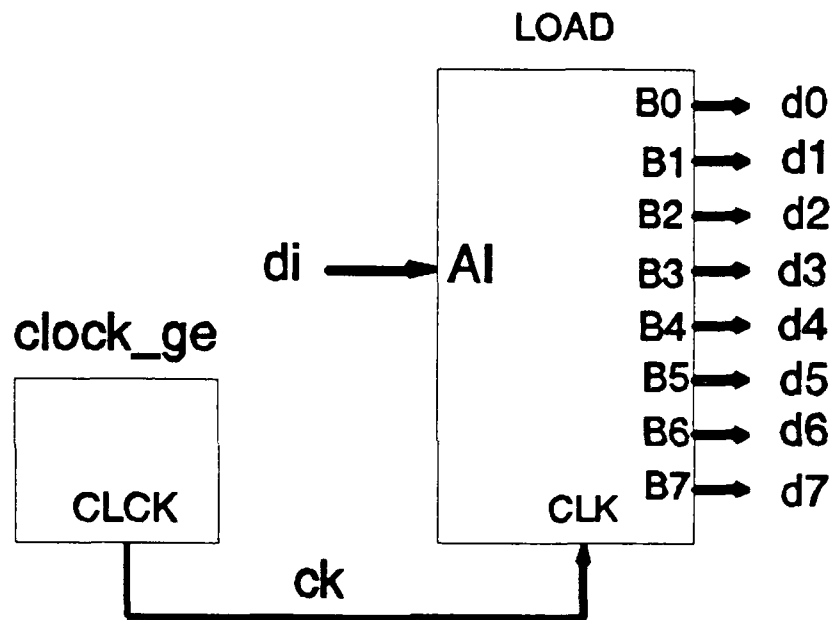


Fig. 5 Serial load parallel shift register block diagram

Figure 5 shows the detailed block diagram of the parallel shift register (LOAD).

The source code in VHDL is shown below

```

entity LOAD is
port (AI : in bit_vector(15 downto 0); B0,B1,B2,B3,B4,B5,B6,B7 :
      out bit_vector(15 downto 0);CLK : in bit);
end LOAD;
architecture BEH of LOAD is

```



```

    type shift is array (0 to 7) of bit_vector(15 downto 0);
begin
    process
        variable A : shift;
        variable I,count : integer := 0;
    begin
        wait until CLK'event and CLK = '1'; -- Clock controls the timing
        for count in 0 to 7 loop
            wait until CLK'event and CLK = '1';
            for I in 0 to 6 loop -- Push input values down to correct position
                A(I) := A(I+1);
            end loop;
            A(7) := AI;
            if (count = 7) and (CLK'event and CLK='1') then -- Output data
                B0 <= A(7);
                B1 <= A(6);
                B2 <= A(5);
                B3 <= A(4);
                B4 <= A(3);
                B5 <= A(2);
                B6 <= A(1);
                B7 <= A(0);
            end if;
        end loop;
        wait on AI,CLK; -- Process activated when sensitivity signal changes
    end process;
end BEH;

```

The input 16-bit data come from AI column by column. The speed of the input data is controlled by the test bench. Note that the first data that appears is the 8th pixel value of the first column. In other words, the sequential order of the incoming data is 7, 6, 5,... 0. In this order, the data is pushed down into the correct position, and the 1-D DCT can be done correctly. After the 1-D DCT computation in Figure 3, the corresponding spectral coefficients will be put back in the correct order, i.e., 0, 1, 2,... 7. "LOAD" module parallel outputs the data to the second circuit "SHIFT" after eight clock cycles (count = 7). After that, it processes another new column of data.

E. SHIFT-TWO-REGISTER MODEL (SHIFT).

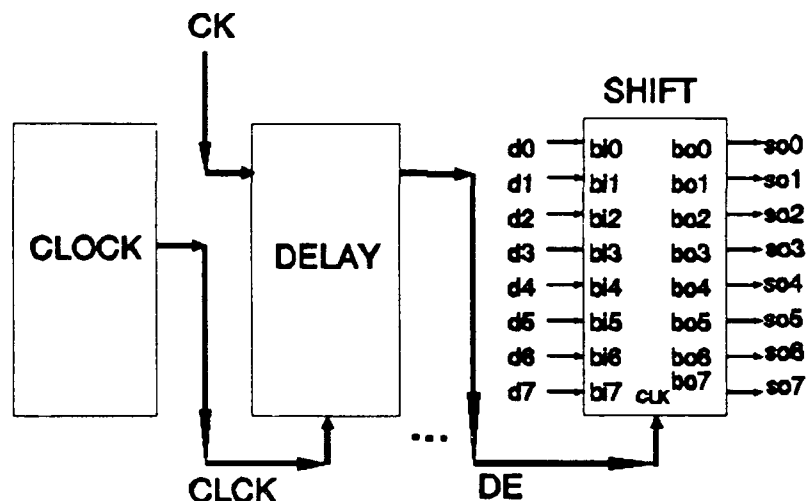


Fig. 6 Shift two register block diagram

The block diagram for SHIFT is shown in Figure 6. There is the second clock generator with three delay gates. Since the incoming pixel values pass through the parallel shift register (LOAD), and it causes a delay of one clock cycle, it is necessary to compensate for this latency by delaying the clock which triggers the shift-two-register (SHIFT). Another clock which runs twice as fast as ck has been used to trigger the original clock passing through the delay line. The VHDL source code of this faster clock is similar to the previously discussed clock generator except the switching period is twice as fast. The assertion time for termination is therefore twice as long. the delay line

consists of shift registers. The VHDL source code of the DELAY and the shift register is as follows

```
entity delay is
  port(a : bit;b : out bit;CLK : bit); --Normal clock coming in from port
a
end delay;
architecture beh of delay is
begin
  process
    variable x : bit;
  begin
    wait until CLK'event and CLK = '1'; -- Faster clock controls timing
    x := a; -- Shifting the incoming clock
    b <= x;
    wait on CLK,a;
  end process;
end beh;
```

```
entity shift is
  port(bi0,bi1,bi2,bi3,bi4,bi5,bi6,bi7 : in bit_vector(15 downto 0);
        bo0,bo1,bo2,bo3,bo4,bo5,bo6,bo7 : out bit_vector(1 downto 0);
        CLK : in bit); -- Port declaration, eight input and output
end shift;
architecture beh of shift is
begin
  process
    variable I : integer := 0; -- counter as well as index
  begin
    for r in 0 to 7 loop
      wait until CLK'event and CLK = '1';
      bo0(0) <= bi0(I); -- "q" = 0 binary weight
      bo0(1) <= bi0(I+1); -- "q" = 1 binary weight
      bo1(0) <= bi1(I);
      bo1(1) <= bi1(I+1);
      bo2(0) <= bi2(I);
      bo2(1) <= bi2(I+1);
      bo3(0) <= bi3(I);
      bo3(1) <= bi3(I+1);
      bo4(0) <= bi4(I);
      bo4(1) <= bi4(I+1);
```

```

        bo5(0) <= bi5(I);
        bo5(1) <= bi5(I+1);
        bo6(0) <= bi6(I);
        bo6(1) <= bi6(I+1);
        bo7(0) <= bi7(I);
        bo7(1) <= bi7(I+1);
        I := I + 2; -- increment of two
    end loop;
    I := 0; -- reset the counter for next column of data
    wait on CLK,bi0,bi1,bi2,bi3,bi4,bi5,bi6,bi7; -- wait for new data
end process;
end beh;

```

The data are input to the shift register in 16-bit words and output in 2-bit words. Note that the counter "I" has been used as an index for each data word. Therefore, a reset ($I := 0$) is necessary after each column of words are done. Otherwise, the index would be running out of range, giving a run time error in the VHDL simulation.

F. 2-BIT ADDER/SUBTRACTOR MODEL (ADDSUB)

The 2-bit adder/subtractor module is shown in Figure 7. The "adsu" VHDL source

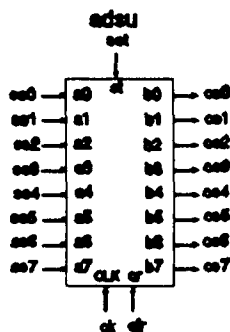


Fig. 7 2-bit add/sub block diagram

code is shown in Appendix A. A simple flow chart in Figure 8 shows the behavior described in VHDL. There are eight 2-bit words input into this circuit. It is necessary to do the "serial" 2-bit addition or subtraction according to the expanded Eqs. (30) and (31). Since the incoming data have been presented in

2's complement notation, 2's complement addition or subtraction should be used. On the other hand, the 2-bit serial operation should consider carriers generated previously. In other words, the first 2-bit addition/subtraction might generate a carrier. This carrier must carry on to the

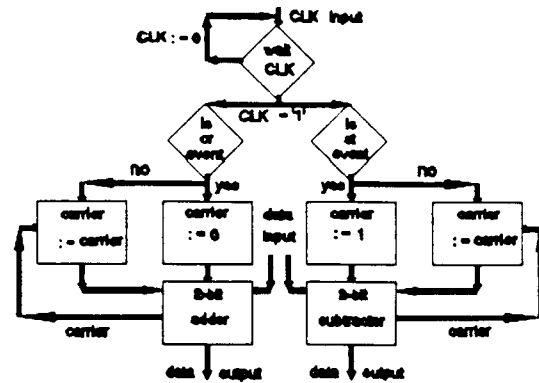


Fig. 8 "adsu" flow chart

next 2-bit add/sub computation. The simplest way to solve this problem is using a 2-bit adder accompanied by a register handing the carrier bit for the next addition/subtraction. For the subtraction case, it is necessary to convert the subtrahend into 2's complement notation and then use the same 2-bit adder to accomplish the computation. What has been done here is to convert the subtrahend into 1's complement first and then add it to "1" at the very first subtraction. The incoming subtrahend is just converted into 1's complement notation and the adder takes care of the "1" addition. In this way, the serial subtraction is accomplished. There are four 2-bit adders and four 2-bit subtractors in the source code. The "cr" bit sets the adder carry at the beginning to zero and the "st" bit sets the subtractor carry to "1". Later on, the adder/subtractor will take care of the carry by itself. For the convenience of notation, the incoming two 2-bit data and the carrier bit have been combined into a 5-bit word, and the addition is done in the 2-bit adder block. There will be more explanation as to how the 2-bit adder block is formed in the later discussion.

G. SHIFT REGISTER

MODEL (REG)

The shift register block diagram is shown in Figure 9. Signal is input from port a and output to port b. The shift register model (REG) VHDL source code is shown below

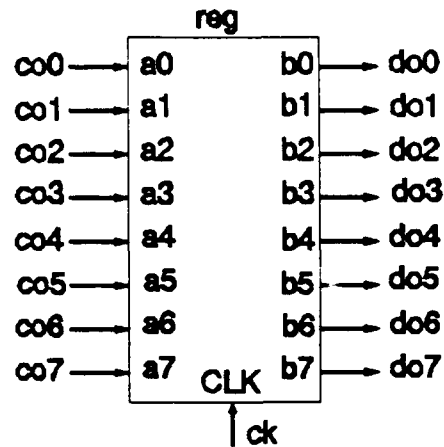


Fig. 9 shift register (reg) block diagram

```
entity reg is
port(a0,a1,a2,a3,a4,a5,a6,a7 : bit_vector(1 downto 0); -- input port
      b0,b1,b2,b3,b4,b5,b6,b7 : out bit_vector(1 downto 0); -- output port
      CLK : bit);
end reg;
architecture beh of reg is
begin
process
variable d0,d1,d2,d3,d4,d5,d6,d7 : bit_vector(1 downto 0);
begin
d0 := a0; -- Substitute the input signal in a variable
d1 := a1;
d2 := a2;
d3 := a3;
d4 := a4;
d5 := a5;
d6 := a6;
d7 := a7;
wait until CLK'event and CLK = '1'; -- Clock control
b0 <= d0; -- shift the variable to output signal
b1 <= d1;
b2 <= d2;
b3 <= d3;
b4 <= d4;
b5 <= d5;
```

```

        b6 <= d6;
        b7 <= d7;
        wait on CLK;
    end process;
end beh;

```

This circuit is the simplest one. The only effect of this code is to use a signal assignment statement to simulate a signal buffer causing a latency period of one clock cycle. The "wait until CLK'event and CLK = '1';" statement activates the timing control. The "wait on CLK" statement activates the process's operation whenever the clock changes its state.

H. READ ONLY MEMORY MODEL (ROM)

Figure 10 shows the read only memory block diagram . The VHDL source code is included in Appendix A. There are eight 2-bit words input to this block, and sixteen 16×16 words corresponding to the 1-D DCT multiplication coefficients being read out. The outputs of four adders with binary weight $q = 0$'s and $q = 1$'s bits form two 4-bit address bus to access the corresponding ROM multiplication coefficients. The same situation happens for subtraction. There are sixteen individual ROM locations with sixteen different values stored in them. Why there are sixteen ROM locations, and why there are sixteen different values stored in them are discussed in detail in later sections. Note that in the address assignment part of the source code, the order of the addresses starts from e0, e1, e2, e3 and ends with e7, e6, e5, e4. This detailed explanation will also be given in later discussion. The values stored in the individual ROM have been

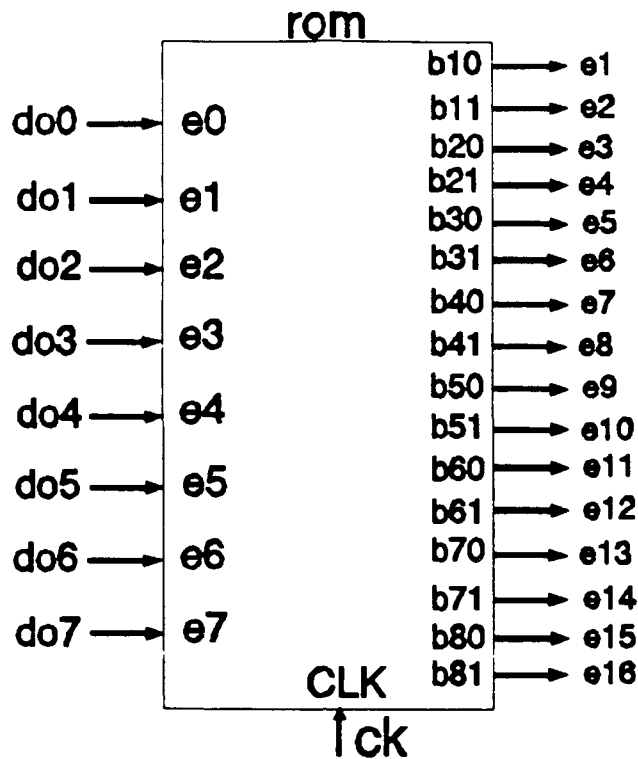


Fig. 10 ROM block diagram

converted from the sum of coefficients " C_{mk} " to 16-bit 2's complement binary values. The values of " C_{mk} " are calculated according to Eq. (15) and Eq. (16).

I. SHIFT RIGHT 1-BIT REGISTER MODEL (SHI_1)

Figure 11 shows the shift right 1-bit register block diagram. Its VHDL source code is included in Appendix A. The shift right 1-bit register receives sixteen 16-bit words and makes the right shift operation in eight words. It outputs the resultant sixteen 16-bit words to the next circuit. The only difference between the input and the output values is that the odd numbered 16-bit words have been shifted right 1 bit position. At the same time, the original 16th bit (sign bit) of each odd word has been checked and replaced by

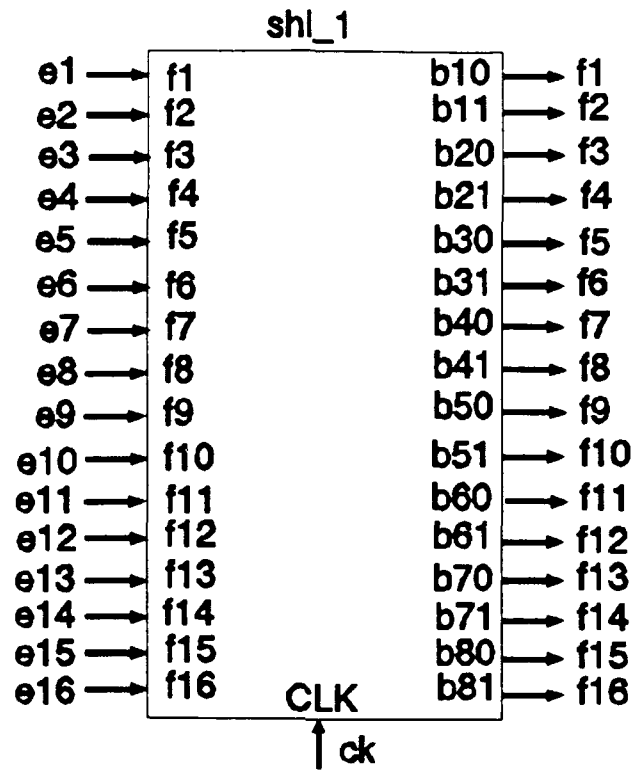


Fig. 11 Shi_1 register block diagram

a proper bit ("0" or "1", depending on whether it has a positive or negative value) to properly extend the binary 2's complement number.

J. ADDER/SUBTRACTOR-G MODEL (ADD_G)

Figure 12 shows the add_g block diagram. It includes one control circuit and five delay gates. The control circuit enables the add_g to do addition or subtraction. The purpose of the delay line is to compensate for signal latency. To activate the add/subtract controller at the right time when signal arrives is a required procedure.

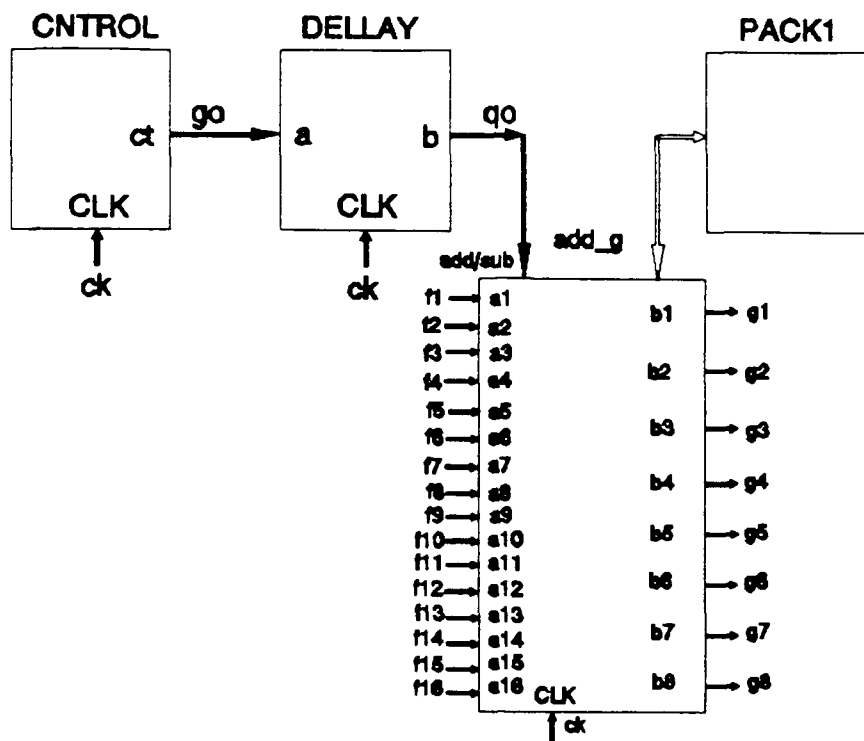


Fig. 12 Add_g block diagram

The add_g VHDL source code as well as the control and the delay VHDL source code are shown below.

```

entity control is
    port(CLK : bit;ct : out bit);
end control;
architecture beh of control is          -- control
begin
    process
        variable i : integer := 0;
    begin
        wait until CLK'event and CLK = '1'; -- Clock triggers the circuit
        if i = 7 then
            ct <= '1'; -- output '1' every eight clock period
        else
            ct <= '0';
        end if;
        i := i + 1;
    end process;
end architecture;

```

```

        if i = 8 then
            i := 0; -- Reset the counter
        end if;
    end process;
end beh;

```

```

entity delay10 is
    port(a : bit;b : out bit;CLK : bit);
end delay10;
architecture beh of delay10 is          -- delay
begin
    process
        variable x : bit;
    begin
        wait until CLK'event and CLK = '1';
        x := a;
        b <= x;
        wait on CLK,a;
    end process;
end beh;

```

```

use work.pack1.all; -- All the functions in pack1 are used
entity add_g is
    port(a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13,a14,a15,a16 :
        bit_vector(15 downto 0); -- input port
        b1,b2,b3,b4,b5,b6,b7,b8 : out bit_vector(15 downto 0); -- output port
        CLK,as : bit);
end add_g;
architecture beh of add_g is
begin
    process
        variable x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,x13,x14,x15,x16,
            n1,n2,n3,n4,n5,n6,n7,n8 : bit_vector(15 downto 0);
        variable y1,y2,y3,y4,y5,y6,y7,y8,y9,y10,y11,y12,y13,y14,y15,y16,
            m1,m2,m3,m4,m5,m6,m7,m8 : integer := 0;
    begin
        wait until CLK'event and CLK = '1';
        x1 := a1; x2 := a2; x3 := a3; x4 := a4; -- input values
        x5 := a5; x6 := a6; x7 := a7; x8 := a8;
        x9 := a9; x10 := a10; x11 := a11; x12 := a12;
        x13 := a13; x14 := a14; x15 := a15; x16 := a16;
        -- Procedure call to do integer conversion
        bi_to_in(x1,y1);bi_to_in(x2,y2);bi_to_in(x3,y3);bi_to_in(x4,y4);
    end process;
end beh;

```

```

        bi_to_in(x5,y5);bi_to_in(x6,y6);bi_to_in(x7,y7);bi_to_in(x8,y8);
        bi_to_in(x9,y9);bi_to_in(x10,y10);bi_to_in(x11,y11);bi_to_in(x12,y12);
        bi_to_in(x13,y13);bi_to_in(x14,y14);bi_to_in(x15,y15);bi_to_in(x16,y16);
        if as = '0' then
            m1 := y1 + y2; m2 := y3 + y4; m3 := y5 + y6; m4 := y7 + y8;
            m5 := y9 + y10; m6 := y11 + y12; m7 := y13 + y14; m8 := y15 +
y16;
        else -- Control gives the subtraction instruction
            m1 := y1 - y2; m2 := y3 - y4; m3 := y5 - y6; m4 := y7 - y8;
            m5 := y9 - y10; m6 := y11 - y12; m7 := y13 - y14; m8 := y15 - y16;
        end if;
        -- Procedure call to do binary conversion
        in_to_bi(m1,n1); in_to_bi(m2,n2); in_to_bi(m3,n3); in_to_bi(m4,n4);
        in_to_bi(m5,n5); in_to_bi(m6,n6); in_to_bi(m7,n7); in_to_bi(m8,n8);
        b1 <= n1; b2 <= n2; b3 <= n3; b4 <= n4;
        b5 <= n5; b6 <= n6; b7 <= n7; b8 <= n8;
        wait on a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13,a14,a15,a16,CLK;
    end process;
end beh;

```

The control is triggered by the clock, and an output of the control bit "ct" is generated. On the 8th clock period, the "ct" becomes "1" but equals "0" otherwise. The delay is also triggered by the clock. It receives one bit and outputs the same bit one clock cycle later.

Add_g has sixteen 16-bit word inputs and eight 16-bit word outputs. It performs 16-bit addition or subtraction. As discussed previously, it is rather complicated to build up a 16-bit adder/subtractor in a VHDL structural approach. The easiest way is to convert the 16-bit binary words into integers. In this way, "use work.pack1.all" at the beginning of the entity has to be declared, in order to call the "bi_to_in" procedure in pack1. "Work" represents the working library used, and "pack1.all" represents all the packages being used. After the conversion of binary values to integer values, addition or subtraction was done according to the control input "as". The results then are converted

back to binary values again for output. Of course, the timing is always synchronized by the clock.

K. SHIFT REGISTER-H MODEL (REG_H)

The `reg_h` block diagram is shown in Figure 13. It functions just like "reg", except "reg" handles 2-bit words and "reg_h" handles 16-bit words. The VHDL source codes are

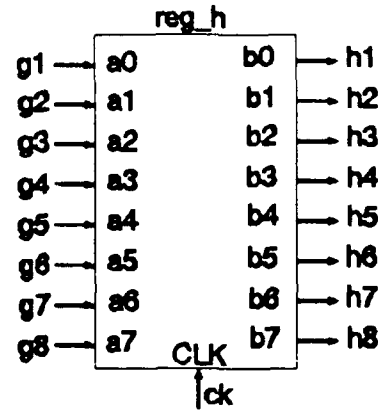


Fig. 13 Shift register_g block diagram

the same except for the declaration of the length of bit-vectors.

L. 16-BIT ADDER_I MODEL (ADD_I)

Figure 14 shows the block diagram of the 16_bit adder (`ADD_I`). `ADD_I` and `ADD_G` are basically the same. `ADD_I` does not have the "as" control bit or "if" instruction in the VHDL source code to do the subtraction. Another big difference is that `ADD_I` is not triggered by the clock. It adds up the two 16-bit inputs with no delay. It does integer addition with the procedures in `pack1` also. The two inputs come from `REG_H` and the feedback output from the `SHI_2`, which shifts the result to the right by 2 bits. This is shown in Figure 2. The VHDL source code for `ADD_I` is shown below

```
use work.pack1.all;
entity add_i is
  port(a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13,a14,a15,a16 :
        bit_vector(15 downto 0);
        b1,b2,b3,b4,b5,b6,b7,b8 : out bit_vector(15 downto 0));
```

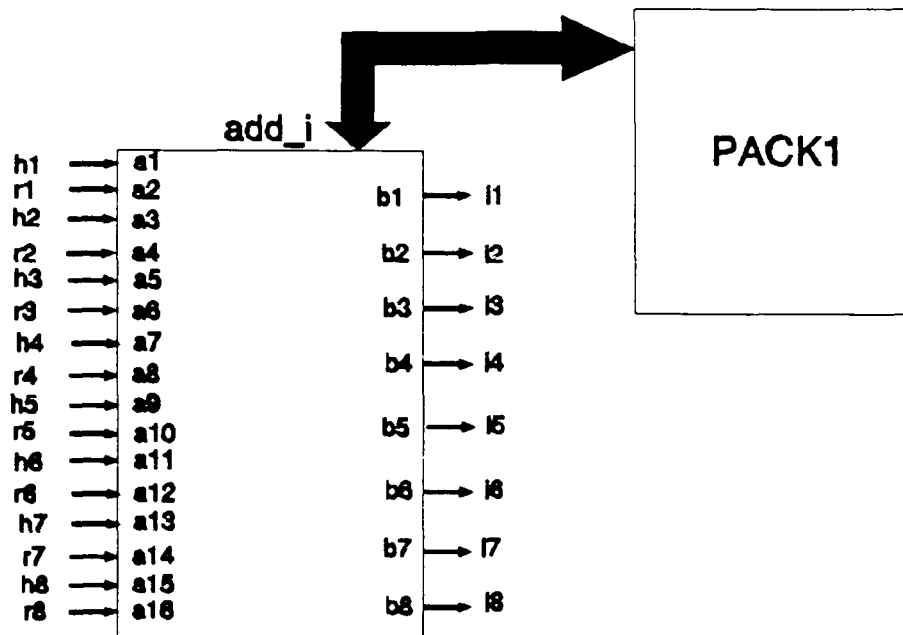


Fig. 14 16-bit add_i block diagram

```

end add_i;
architecture beh of add_i is
begin
  process
    variable x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,x13,x14,x15,x16,
             n1,n2,n3,n4,n5,n6,n7,n8 : bit_vector(15 downto 0);
    variable y1,y2,y3,y4,y5,y6,y7,y8,y9,y10,y11,y12,y13,y14,y15,y16,
             m1,m2,m3,m4,m5,m6,m7,m8 : integer := 0;
  begin
    x1 := a1; x2 := a2; x3 := a3; x4 := a4;
    x5 := a5; x6 := a6; x7 := a7; x8 := a8;
    x9 := a9; x10 := a10; x11 := a11; x12 := a12;
    x13 := a13; x14 := a14; x15 := a15; x16 := a16;
    bi_to_in(x1,y1);bi_to_in(x2,y2);bi_to_in(x3,y3);bi_to_in(x4,y4);
    bi_to_in(x5,y5);bi_to_in(x6,y6);bi_to_in(x7,y7);bi_to_in(x8,y8);
    bi_to_in(x9,y9);bi_to_in(x10,y10);bi_to_in(x11,y11);
    bi_to_in(x12,y12);
    bi_to_in(x13,y13);bi_to_in(x14,y14);bi_to_in(x15,y15);
    bi_to_in(x16,y16);
  end process;
end architecture;

```

```

m1 := y1 + y2; m2 := y3 + y4; m3 := y5 + y6; m4 := y7 + y8;
m5 := y9 + y10; m6 := y11 + y12; m7 := y13 + y14; m8 := y15
+ y16;

in_to_bi(m1,n1); in_to_bi(m2,n2); in_to_bi(m3,n3); in_to_bi(m4,n4);
in_to_bi(m5,n5); in_to_bi(m6,n6); in_to_bi(m7,n7); in_to_bi(m8,n8);
b1 <= n1; b2 <= n2; b3 <= n3; b4 <= n4;
b5 <= n5; b6 <= n6; b7 <= n7; b8 <= n8;
wait on a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13,a14,a15,a16;
end process;
end beh;

```

M. SHIFT RIGHT 2-BIT REGISTER MODEL (SHI_2)

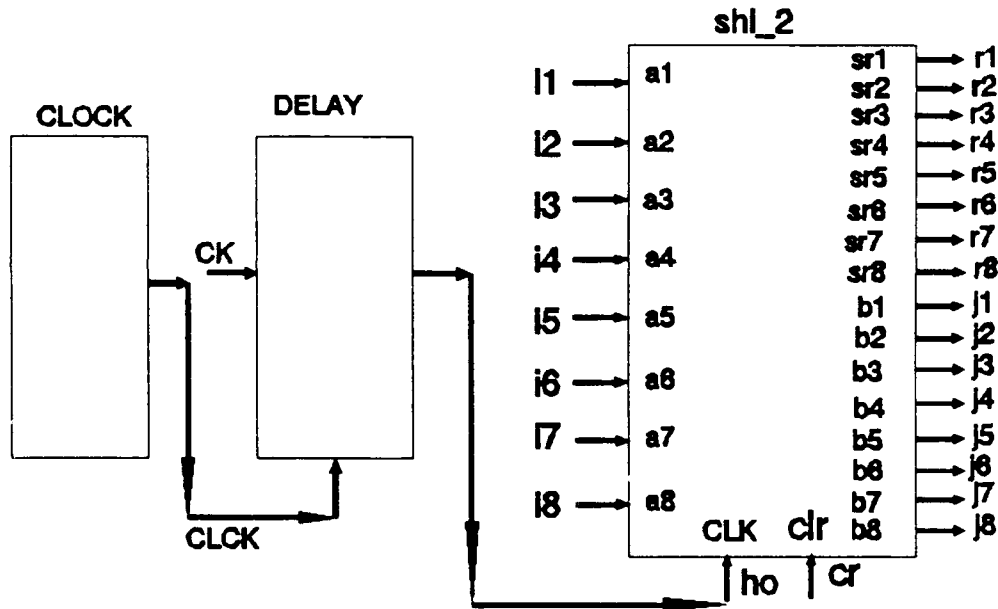


Fig. 15 Shift right 2-bit register block diagram

The shift right 2-bit register (shi_2) block diagram is shown in Figure 15. It includes another clock generator running two-times faster to trigger the delay unit which delays the normal clock by one period. It has another clear line (clr) from the test bench

that clears the register every eight clock cycles. The VHDL source code of SHI_2 is shown in Appendix B.

The SHI_2 model has eight 16-bit word inputs from ADD_I and has sixteen 16-bit word outputs. The input values have been checked for the sign bit, and the SHI_2 shifts the data 2 bits to the right in proper 2's complement representation. There are eight blocks in the SHI_2 module. The results are updated and fed back to ADD_I module to perform an addition with the incoming data values. In every 8th clock cycle, the results are parallel shifted to the "parallel load serial shift" register (RESULT). During the same cycle, the shift right 2-bit results are cleared, and the SHI_2 is ready for the next column operation.

N. PARALLEL LOAD SERIAL SHIFT REGISTER MODEL (RESULT)

The block diagram of the parallel load serial shift register (RESULT) is shown in Figure 16. There are eight inputs from SHI_2; RESULT puts out only one value at a time. The VHDL source code of RESULT is shown below,

```
entity result is
  port(a1,a2,a3,a4,a5,a6,a7,a8 : bit_vector(15 downto 0);
        k : out bit_vector(15 downto 0);CLK : bit);
end result;
architecture beh of result is
  type r is array (0 to 7) of bit_vector(15 downto 0);
  begin
    process
      variable x : r;
    begin
      x(0) := a1; x(1) := a2; x(2) := a3; x(3) := a4;
      x(4) := a5; x(5) := a6; x(6) := a7; x(7) := a8;
      for i in 0 to 7 loop
        wait until CLK'event and CLK = '1';
```

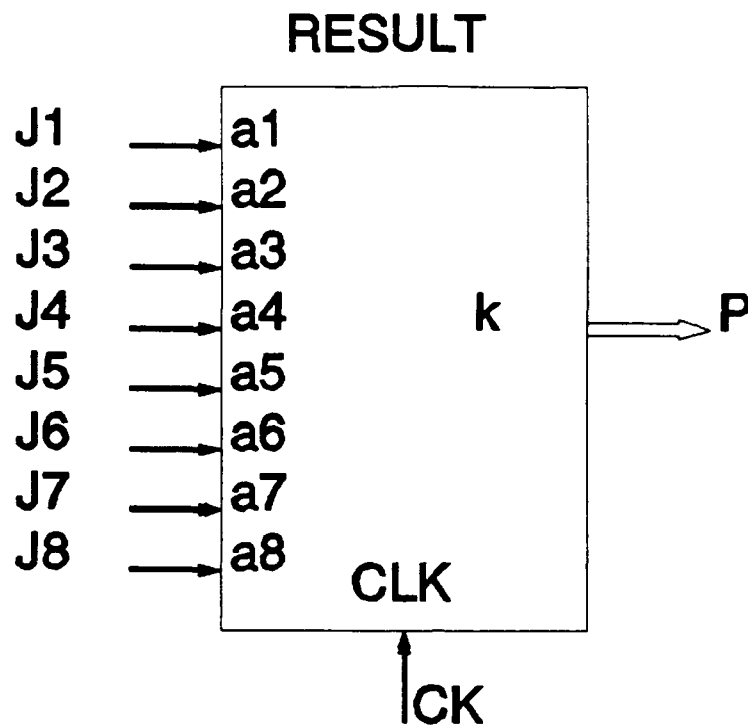



Fig. 16 Parallel shift serial output register block diagram

```

    k <= x(i);
  end loop;
  wait on a1,a2,a3,a4,a5,a6,a7,a8,CLK;
end process;
end beh;

```

Eight 16-bit words are input into RESULT every 8th clock cycle. They are pushed out one value at a time at every clock period. After all eight values have been output, new values are fed in again for the next cycle.

O. TEST BENCH

The Test bench block diagram is shown in Figure 17. It actually includes all the intermediate signals, the control signals, and the input and output signals. The VHDL source code for the test bench is shown in Appendix B. All the components used

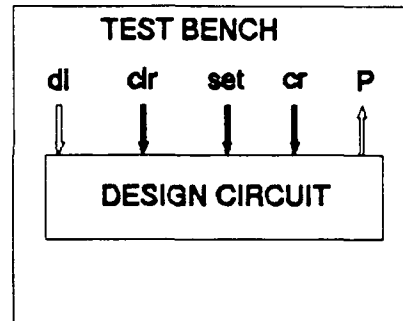


Fig. 17 Block diagram of Test Bench

in the system have been declared and instantiated. The signals used for the simulation are declared also. Configuration statement binds all the components to the test system. The input pixel values are fed into the system through "di", and it is simulated. The results of the simulation are collected by signal "p". A table of the simulation results "p" is generated and analyzed to see if the design is functioning correctly.

V. SIMULATION OUTPUT ANALYSIS AND EXPERIENCE

A. FORMATION OF ROM STORAGE VALUES

As discussed before, there are only sixteen-word ROM for each multiplication coefficient due to the symmetry in DCT. The coefficients can be calculated according to Eq. (15) and Eq. (16).

Table I: Multiplication Coefficients

$m = 0$	$m = 1$	$m = 2$	$m = 3$	
$C_{mk}, k = \text{even}$				
$A = Y_{i0} + Y_{i7}$	$B = Y_{i1} + Y_{i6}$	$C = Y_{i2} + Y_{i5}$	$D = Y_{i3} + Y_{i4}$	U
.3535533905	.3535533905	.3535533905	0.3535533905	$k = 0$
.4619397662	.1913417161	-.1913417161	-.4619397662	$k = 2$
.3535533905	-.3535533905	-.3535533905	.3535533905	$k = 4$
.1913417161	-.4619397662	.4619397662	-.1913417161	$k = 6$
$C_{mk}, k = \text{odd}$				
$A = Y_{i0} - Y_{i7}$	$B = Y_{i1} - Y_{i6}$	$C = Y_{i2} - Y_{i5}$	$D = Y_{i3} - Y_{i4}$	V
.4903926402	.4157348061	.2777851165	.0975451610	$k = 1$
.4157348061	-.097545161	-.4903926402	-.2777851165	$k = 3$
.2777851165	-.4903926402	.0975451610	.4157348061	$k = 5$
.0975451610	.2777851165	.4157348061	-.4903926402	$k = 7$

Since $N = 8$, the expanded equation of Eq. (30) and Eq. (31) can be derived as in Table I after substituting the proper index (m, k). The labels U0, U2, ..., V7 are included in

the table for better understanding. Labels A, B, C, D stand for bit patterns. For example, if $A = 1$, $B = 0$, $C = 1$, $D = 1$, then the values in column 1, 3, and 4 should be summed up to get the corresponding multiplication coefficient sum stored in the ROM. The bit pattern in the circuit has two weighted groups (LSB group $q = 0$'s, and MSB group $q = 1$'s). The coefficient values for these two patterns are exactly the same. Therefore, there are only $8 \times 16 = 128$ different coefficient sums stored in ROM.

One very important fact must be stressed. Are the values stored in the ROM decimal numbers? The answer is obviously no. The values are stored in the ROM as binary numbers. How can these summed decimal numbers be converted into binary numbers? Upon inspection of Table I, it is noted that the largest possible decimal number generated is not greater than 2. The smallest possible decimal number generated is not lesser than -2. As stated before, the number system used here is 16-bit 2's complement number. Therefore, one sign bit, one digit bit, and fourteen fraction bits are chosen to represent the binary numbers stored in the ROM. All the decimal coefficients calculated according to the specific bit pattern A, B, C, D have to be converted into binary 2's complement 16-bit numbers. This conversion operation is carried out with the help of a small program written in Matlab listed in Appendix C. The actual values stored in the ROM are shown in the ROM VHDL source code.

B. SIMULATION AND TESTING IMAGE PATTERN (I)

The first image pattern being used is shown in Figure 17. It is a two-dimensional cosine wave with intensity varied along x -axis. The pixel value can be represented in 128

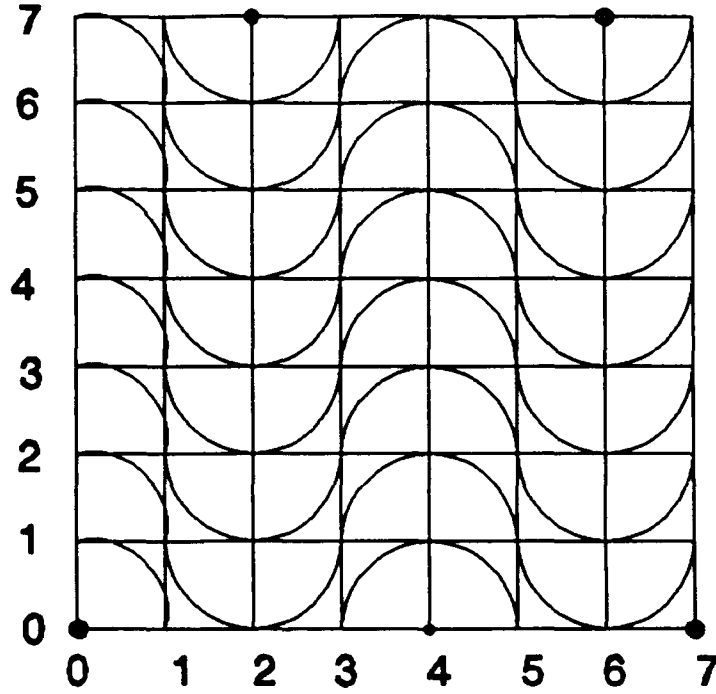


Fig. 18 Pattern (I) 8×8 image block

levels. Therefore, the pixel value of each point in this image can be represented from the following formula

$$f(x, y) = [\cos(2\pi f_x x + 2\pi f_y y) + 1] / 2 \times 128 \quad (32)$$

where $f_x = 1/4$, $f_y = 0$.

After substituting the corresponding index (x, y) in Figure 17 into Eq. (32), the pixel values represented in this 8×8 image block can be shown in Table II. The 12-bit binary representations of decimal numbers 128 and 64 are "000010000000" and "000001000000". Converting the values in Table II into 12-bit binary numbers and taking

Table II: 8×8 image pixel values of Pattern (I)

$y = 7$	128	64	0	64	128	64	0	64
$y = 6$	128	64	0	64	128	64	0	64
$y = 5$	128	64	0	64	128	64	0	64
$y = 4$	128	64	0	64	128	64	0	64
$y = 3$	128	64	0	64	128	64	0	64
$y = 2$	128	64	0	64	128	64	0	64
$y = 1$	128	64	0	64	128	64	0	64
$y = 0$	128	64	0	64	128	64	0	64
	$x = 0$	$x = 1$	$x = 2$	$x = 3$	$x = 4$	$x = 5$	$x = 6$	$x = 7$

them column by column into the 1-D DCT VHDL model yields the corresponding 1-D DCT spectral

coefficients (in Hex) as listed in Table III. The same decimal values in Table II has also been put into a 1-D DCT subroutine for calculation which is in a image processing library called spider. The result is shown in Table IV.

Due to the time limitations, the attempt to carry out the transpose of the 1-D DCT coefficients in VHDL behavior models was not made. However, manual transpose is done instead. Transposed 1-D DCT coefficients of pattern (I) in VHDL simulation is shown in Table V. The values in Table V are converted again into binary numbers and

Table III: 1-D DCT spectral coefficients of Pattern (I) in VHDL simulation

0B50	05A8	0000	05A8	0B50	05A8	0000	05A8
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000

Table IV: 1-D DCT coefficients of pattern (I) using Spider Subroutine

362.03	181.01	0	181.0 1	362.03	181.01	0	181.01
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

input column by column into the 16-bit 1-D DCT VHDL model to accomplish the 2-D DCT operation. The 2-D DCT spectral coefficients which have been transposed back in the VHDL simulation are shown in Table VI. The 1-D DCT operations in the VHDL simulation is based on integer calculation. In order to prove that the 1-D DCT VHDL

Table V: Transposed 1-D DCT coefficients of pattern (I) in VHDL simulation

0B50	0000	0000	0000	0000	0000	0000	0000
05A8	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	00000
05A8	0000	0000	0000	0000	0000	0000	0000
0B50	0000	0000	0000	0000	0000	0000	0000
05A8	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
05A8	0000	0000	0000	0000	0000	0000	0000

Table VI: 2-D DCT spectral coefficients of pattern (I) in VHDL simulation

01F7	005F	0000	00C4	00FF	FF7C	0000	FFED
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000

simulation result is correct, the values in Table V are converted into integers and are shown in Table VII. The values in Table VII is again calculated column by column using the spider 1-D DCT subroutine. Its 2-D DCT spectral coefficients are transposed and shown in Table VIII.

Table VII: Table V in integer values

2896	0	0	0	0	0	0	0
1448	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1448	0	0	0	0	0	0	0
2896	0	0	0	0	0	0	0
1448	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1448	0	0	0	0	0	0	0

Table VIII: 2-D DCT spectral coefficients of pattern (I) using Spider Subroutine

4095.5	768.54	0	1573.0	2047.7	-1051.0	0	-152.8
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

To ensure that the 1-D DCT structural calculation in the VHDL simulation is correct , direct 1-D DCT calculation on a calculator is also carried out based on Eq.(15), and Eq.(16). Equations (33) and (34) show the calculation example for $k = 0$ and $k = 1$.

$$C(0) = \frac{1}{\sqrt{8}}(2896+1448+0+1448+2896+1448+0+1448) \quad (33)$$

$$C(1) = \sqrt{\frac{2}{8}}(2896\cos\frac{\pi}{16} + 1448\cos\frac{3\pi}{16} + 0 + 1448\cos\frac{7\pi}{16} + 2896\cos\frac{9\pi}{16} + 1448\cos\frac{11\pi}{16} + 0 + 1448\cos\frac{15\pi}{16}) \quad (34)$$

The results using this approach are listed in Table IX. Note that the results of Table VIII and Table IX are very close.

Table IX: 2-D DCT coefficients of pattern (I) using direct calculation

4095.5	768.59	0	1537.0	2047.7	-1051.0	0	-152.8
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

It is also necessary to trace the operation in the VHDL structural models shown in Figure 2. To understand the structural operation and calculation of the 1-D DCT in the VHDL simulation in more detail, a manual derivation and calculation are carried out for

Table X 16-bit binary number representation of table (V)

Slice(0)	0000101101010000	0	0	0	0	0	0	0
Slice(1)	0000101101010000	0	0	0	0	0	0	0
Slice(2)	0000000000000000	0	0	0	0	0	0	0
Slice(3)	0000010110101000	0	0	0	0	0	0	0
Slice(4)	0000101101010000	0	0	0	0	0	0	0
Slice(5)	0000010110101000	0	0	0	0	0	0	0
Slice(6)	0000000000000000	0	0	0	0	0	0	0
Slice(7)	0000010110101000	0	0	0	0	0	0	0

the purpose. First the values in Table V need to be converted into binary numbers, which are shown in Table X. It is clear that only one column of Table X is not zero. Therefore, there is only one column of the 1-D DCT that needs computation. The values in the first column are input into the 1-D DCT VHDL model which yields the serial 2-bit addition/subtraction results as shown in Table XI.

The first column in Table XI shows how the 2-bit addition/subtraction is done. The first row on the top represents the clock cycle. The rows in the upper-half (U) correspond "k" equal to even numbers, and the rows in the lower-half (V) correspond "k" equal to odd numbers. Each half column has four bits, forming a bus to address the corresponding ROM coefficients. For example, at the first clock cycle, there are two 4-bit buses. The four least significant bits (LSB) form an "ABCD" corresponding to "0000" bus to address the "U00" (refer to Fig. 2) ROM value. This yields the value "0000000000000000" as output. The MSBs of the first clock cycle addresses the "U01" ROM value

Table XI: Serial 2-bit addition/subtraction output

	clock=8	clock=7	clock=6	clock=5	clock=4	clock=3	clock=2	clock=1		
Slice (0+7)	00	01	00	00	11	11	10	00	A	U
Slice (1+6)	00	00	01	01	10	10	10	00	B	
Slice (2+5)	00	00	01	01	10	10	10	00	C	
Slice (3+4)	00	01	00	00	11	11	10	00	D	
Slice (3-4)	11	11	01	10	01	01	10	00	D	V
Slice (2-5)	11	11	01	10	01	01	10	00	C	
Slice (1-6)	00	00	10	01	10	10	10	00	B	
Slice (0-7)	00	00	10	01	10	10	10	00	A	

"0000000000000000" out. It then adds up with the 1-bit right shifted value of "U00". This result is stored in REG_H and then 2-bit right-shifted in the SHI_2 register. The first clocked 2-bit right-shifted word is then fed back to ADD_I and added to the second clocked result "0101101010000010". The procedure of getting this second clocked result is just the same as that of getting the first clocked result. The summation of the first 2-bit right-shifted number and the second clocked result "0101101010000010" is then shifted right 2 bits, yielding "0001011010100000". This value is then added to the third clocked result "0111000100100010", yielding "1000011111000010". This process goes on serially until the 8th clock cycle is reached. The addressed output ROM value of the MSB of the 8th clock cycle "0000000000000000" is subtracted from the right-shifted 1-bit

Table XII 2-D DCT coefficients of pattern (I) using manual calculation

0000000111111111	0000000001011111	0000000000000000	0000000011000100	0000000111111111	1111111011111100	0000000000000000	111111111101100
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

addressed ROM value of the LSB of the 8th clock cycle "0000000000000000". This result is then added to the previous accumulated 7 clocked values, yielding "0000011111111111". This final result is then right shifted 2 bits, yielding "0000000111111111" and output as the first pixel 2-D DCT coefficient of the first column. 8×8 image block of the 2-D DCT coefficients pattern I using structural manual calculations are shown in Table XII. The detailed calculation procedure is listed in Appendix D. Note that the summation of the accumulated two clocked values and the third clocked result generates an overflow. This overflow will eventually generate a negative value when right-shifted 2 bits. This is a inherent drawback of using 16-bit integers arithmetics.

C. SIMULATION AND TEST OF IMAGE PATTERN (II)

Image pattern II is equal to image pattern I rotated by 45°. The following formula was used to calculate each pixel value.

$$f(x,y) = [\cos(2\pi(\frac{1}{4T})Tx + 2\pi(\frac{1}{4T})Ty) + 1] / 2 \times 128 \quad (35)$$

Table XIII: 8 × 8 image block pixel values of pattern (II)

7	64	128	64	0	64	128	64	0
6	0	64	128	64	0	64	128	64
5	64	0	64	128	64	0	64	128
4	128	64	0	64	128	64	0	64
3	64	128	64	0	64	128	64	0
2	0	64	128	64	0	64	128	64
1	64	0	64	128	64	0	64	128
0	128	64	0	64	128	64	0	64
	0	1	2	3	4	5	6	7

The 8 × 8 image block pixel values of pattern II represented in decimal numbers are shown in Table XIII. The 2-D DCT of pattern II has been calculated in two ways, VHDL simulation and spider subroutine. Using VHDL simulation first, Table XIII is converted into binary numbers and is input column by column into the VHDL 1-D DCT test bench. Its 1-D DCT coefficients is shown in Table XIV. For 2-D DCT, the values

Table XIV: 1-D DCT coefficients of pattern (II) using VHDL simulation

7	016A	016A	016A	016A	016A	016A	016A	016A
6	FFBB	0043	0043	FFBB	FFBB	0043	0043	FFBB
5	0000	0000	0000	0000	0000	0000	0000	0000
4	FF74	008A	008A	FF74	FF74	008A	008A	FF74
3	00B5	00B5	FF4A	FF4A	00B5	00B5	FF4A	FF4A
2	005D	FFA2	FFA2	005D	005D	FFA2	FFA2	005D
1	0000	0000	0000	0000	0000	0000	0000	0000
0	000D	FFF2	FFF2	000D	000D	FFF2	FFF2	000D
	0	1	2	3	4	5	6	7

in Table XIV are then transposed manually, and the results are input into the 16-bit VHDL 1-D DCT test bench. The 2-D DCT spectral coefficients for pattern II in VHDL simulation are listed in Table XV.

Table XV: 2-D DCT coefficients of pattern (II) using VHDL simulation

7	005F	0000	0000	0000	0000	0000	0000	0000
6	FFFF	0000	0000	0000	FFE7	0000	0000	0030
5	0000	0000	0000	0000	0000	0000	0000	0000
4	FFFF	0000	0000	0000	FFCE	0000	0000	0000
3	F555	0017	0000	0031	0000	FFDE	0000	FFFC
2	FFFF	0000	0000	0000	0020	0000	0000	0000
1	0000	0000	0000	0000	0000	0000	0000	0000
0	FFFF	0000	0000	0000	0005	0000	0000	0000
	0	1	2	3	4	5	6	7

Table XVI: Pattern II 1-D DCT coefficients using Spider Subroutine

7	181.0	181.0	181.0	181.0	181.0	181.0	181.0	181.0
6	-33.97	33.97	33.97	- 33.97	- 33.97	33.97	33.97	- 33.97
5	0	0	0	0	0	0	0	0
4	-69.53	69.53	69.53	- 69.53	- 69.53	69.53	69.53	- 69.53
3	90.51	90.51	-90.51	- 90.51	90.51	90.51	-90.51	- 90.51
2	46.46	-46.46	-46.46	46.46	46.46	- 46.46	-46.46	46.46
1	0	0	0	0	0	0	0	0
0	6.757	-6.757	-6.757	6.757	6.757	- 6.757	-6.757	6.757
	0	1	2	3	4	5	6	7

1-D DCT subroutine in Spider is used to double check the VHDL simulation result. Values in Table XIII are calculated column by column, and its result is listed in Table XVI. This result is compared with that of Table XIV for verification.

2-D DCT floating point calculation is also used to check the VHDL simulation. Again for the same reason of comparison, values in Table XIV are chosen and converted into integers. After the Hex-integer conversion, these values are transposed again and calculated by 1-D DCT Spider subroutine column by column. The results are shown in Table XVII.

Table XVII: 2-D DCT coefficients of pattern (II) using floating point calculation

7	1023.9	0	0	0	0	0	0	0
6	-2.828	0	0	0	-192.3	0	0	0
5	0	0	0	0	0	0	0	0
4	-2.828	0	0	0	-393.2	0	0	0
3	-1.414	192.7	0	394.4	0	-263.5	0	-38.33
2	-1.414	0	0	0	264.5	0	0	0
1	0	0	0	0	0	0	0	0
0	-1.414	0	0	0	38.18	0	0	0
	0	1	2	3	4	5	6	7

D. RESULT ANALYSIS

There are four methods being used to prove the accuracy of the VHDL structural 1-D DCT in VHDL simulation. Comparing Tables VI, VIII, IX, and XII, the similarities among them are obvious. Tables VIII and IX are almost the same while Tables VI and XII need to be converted into decimal numbers for ease of comparison. Table VI needs to be converted into 16-bit binary values first, then using the definition of the 16-bit binary number system (1 sign bit, 1 integer and 14 fraction bits) to convert the binary words into decimal numbers.

The multiplication factor as to how many times the number is being right-shifted here is 2^{17} . The equivalent integer values of Table VI and Table XII are shown in Table XVIII and XIX. Most of the pixel values are similar to those in Table VIII and IX with a few differences. There are two reasons that can explain this phenomenon. First, there is a limitation in 16-bit binary number representation. Those fractional numbers that are

Table XVIII Equivalent decimal numbers of table (VI)

4024	760	0	1568	2040	-1056	0	-152
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Table XIX Equivalent decimal numbers of table (XII)

4088	760	0	1568	2040	-1056	0	-160
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

smaller than 2^{14} are truncated. This will cause small difference between Table VI,XII and Table XVIII,XIX. The second reason is due to the overflow situation. The accumulated sum of the coefficients might be greater than the biggest number that a 16-bit binary number system could represent. This overflow situation will cause larger difference between Table VI,XII and Table XVIII,XIX.

A way is found to indicate the overflow situation. Checking can be made in ADD_G and ADD_I by adding the following VHDL source code right after the integer to binary number conversion.

```

if ((x1(15) = '1' and x2(15) = '1' and n1(15) = '0') or
    (x1(15) = '0' and x2(15) = '0' and n1(15) = '1')) then
    over1 <= '1';

if ((x3(15) = '1' and x4(15) = '1' and n2(15) = '0') or
    (x3(15) = '0' and x4(15) = '0' and n2(15) = '1')) then
    over2 <= '1';

```

```

if ((x15(15) = '1' and x16(15) = '1' and n8(15) = '0') or
    (x15(15) = '0' and x16(15) = '0' and n8(15) = '1')) then
    over8 <= '1';

```

Of course, at the port declaration, a special signal declaration must be made in order to notify the test bench about this overflow condition. VHDL source code for the port declaration is shown below.

```

port(--;b1,b2,b3,b4,b5,b6,b7,b8 : out bit_vector(15 downto 0);
    over1,over2,over3,over4,over5,over6,over7,over8 : out bit;
    CLK : bit);

```

In addition to the port modification, the test bench component's port also needs to be modified. the last thing to accomplish in signaling this overflow condition is to declare

signals and unable the "port map" to receive the overflow signal coming from ADD_G and ADD_I. VHDL source code is shown below.

```

-----
signal ov1,ov2,ov3,ov4,ov5,ov6,ov7,ov8 : bit;
-----

g : add_g port map(f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,f13,f14,f15,f16,
                  g1,g2,g3,g4,g5,g6,g7,g8,ck,qo,ov1,ov2,ov3,ov4,ov5,ov6,ov7,ov8);
-----

i : add_i port map(h1,r1,h2,r2,h3,r3,h4,r4,h5,r5,h6,r6,h7,r7,h8,r8,
                  i1,i2,i3,i4,i5,i6,i7,i8,ck,ov1,ov2,ov3,ov4,ov5,ov6,ov7,ov8);
-----

```

Whenever the overflow bit "ov#" changes to '1', it indicates that particular pixel value has experienced overflow.

E. EXPERIENCE

My experience in the work can be listed as follows.

1. Input Data Sequential Order error

The sequential order of input pixels which are input to the parallel shift register was assumed to be 7, 6, ...0. According to the transposed sequence, the actual input data should be in the order of 0, 1, 2,...7. Therefore, there would be an error if the sequence of the transposed data is not reversed. This means that another reverse circuit should be added between the transpose circuit and the input "load" circuit. But,

it is rather complicated to add an extra circuit. The easiest way to solve this problem is to input the data in the order of 0, 1,...7 and switch the subtrahend connections (0-7, 1-6, 2-5, 3-4) in the 2-bit adder/subtractor circuit. In this way, the order of input data and output data are always in the order of 0, 1, 2,... 7 and it is not necessary to add an extra circuit.

2. Formation of 2-bit Adder in VHDL source code

The interface of a 2-bit adder has five inputs (two for the adder, two for the addend, and one for the carrier), three outputs (two for the addition result, and one for the carrier). Thus, a truth table involving all possible input combinations can be made. There are five inputs, therefore $2^5 = 32$ combinations will occur. After building up an 8×32 truth table, Karnaugh map reduction can be used to minimize the complex expression in boolean algebra. It is the boolean algebra expression which is used in the VHDL source code. There is a detailed example listed in Appendix F.

3. No Timing control in Add_i Model

Almost every circuit needs a clock to trigger and control the sequential process. ADD_I is a special adder circuit without a triggering clock. As mentioned earlier, the accumulator of the serial bit result consists of ADD_I and SHI_2. ADD_I is used to add up the incoming clocked result with the latest accumulated result right after right-shifting by 2 bits. If these two circuits are triggered by the clock, then there will be a time delay of one clock cycle between ADD_I and SHI_2. In other words, ADD_I is adding the incoming clocked result with the accumulated right-shifted 2-bit result from

one clock cycle earlier, rather than the latest. This will cause an error in the output coefficients. The method to remove of this time delay of one clock cycle between ADD_I and SHI_2 is to allow only one clock to trigger this accumulator. Another alternative considered is to use the clock to trigger ADD_I instead of triggering SHI_2. However, the experiment shows that this cannot be done, since SHI_2 has to be cleared on every 8th clock cycle, and this clearing needs a counter to calculate the exact time. On the other hand, SHI_2 is to output the correct accumulated result every 8th clock period. These two factors both need a clock to control the timing. This is why ADD_I was chosen not to be triggered by the clock.

4. "Set" control in Test Bench

It is strange enough that the "set" control in the test bench does not get the value '1' at the beginning of simulation. The function of "set" is to initiate all the subtractor's carriers in "adsu" to '1' in order to accomplish the subtraction. This initiation is performed only once. The carrier of the subtractor is then carried over all by itself. That is to say, the carrier is a variable in "adsu". This carry variable is initiated by the "set" first and will be influenced by the "set" at subsequent times if modification of the signal "set" is not made. Fortunately, "set" has to change only once from '0' to '1' at the beginning of the simulation. Therefore, an "event" instruction causes "set" to be a sensitivity signal. Since "set" changes only once, it will not have any further influence on the carrier variable. Other than this, the time for "set" to change its state is very important. the clock is '0' at the beginning of the simulation and changes its state to '1' after 5 ns. If "set" changes its state other than at 5 ns, the subtraction result will

be wrong. Only when "set" changes its state at 5 ns will the result of subtraction be correct.

5. Signals cannot be used as variables in VHDL

In solving the problem mentioned in previous section, efforts have been made to use the "set" signal directly as a variable within the process. This certainly will yield a syntax error doing compilation of the source codes.

6. Preventing Negative Zero occurrences in Pack1

There is a paragraph of source code added to pack1 at the end of "in_to_bi" when negative zeros found during the simulation. When these negative zeros arrive at the gate of shi_2, they will generate very large negative numbers and cause an error at the output. This unwanted situation has been taken care of by adding source code to check for negative zeros at the end of the integer-binary conversion procedure. Although this extra checking source code works fine, it means an extra circuit must be added. This is not the goal in circuits design. A close inspection of in_to_bi source code has been made and a very small mistake has been found. At the beginning of inverting the bit stream into 2's complement codes, positive or negative integers is checked in order to assign the correct sign bit "w(15)" for the converted binary number. It is found that "w(15) := '0'" is only assigned to the situation when "m > '0'". The other values are all assigned with "w(15) := '1'". This is how negative zeros are generated. Had the source code "m > '0'" been changed to "m >= '0'", the extra negative zero checking codes would not be necessary.

VI. CONCLUSION

The main objectives of this thesis, using the VHDL to describe a 1-D DCT structural architecture of a 8×8 image block and simulating it on a workstation, have been reached. The basic theory of 1-D DCT, the principle of distributed arithmetic and the actual hardware architecture have been made more clear in the VHDL simulation. Above all, the experience of using the VHDL to describe an algorithm and the simulation of the VHDL is obtained. Although getting familiar with the language and its simulation has been time-consuming, the benefits of the signal tracing and the time modeling have been demonstrated in this thesis. VHDL itself is a portable document and a hierarchical language. Therefore, this thesis can be adopted in other more complicated design.

Despite the fact that the VHDL simulation result of integer point calculation is not as precise as floating point calculation, the resultant energy spectrum of 1-D DCT is already good enough to recover the original image block. Besides, absolute value accuracy is not important for image compression. It is the relative value between pixel points that matters. Another point worthy to mention is that the approach in this thesis has the advantages of calculation speed, since the hardware for floating point calculation is much more complicated than that for integer point calculation.

There is still a very important module that was not described, the transpose module. The transpose module can be connected to the test bench and fulfill the automatic 2-D DCT simulation.

The simulation done here is only the initial part of the "top-down design" process. The algorithm of an 8×8 image block 2-D DCT in VHDL behavior description was implemented. This behavior description can be further developed into gate level descriptions. Once reached the gate level, the hardware circuit implementation can be realized.

APPENDIX A. 12-BIT 1-D DCT VHDL SOURCE CODES

----- Normal clock generator -----

```
entity clock_ge is
  port(CLCK :inout bit);
end clock_ge;
architecture clk_ctl of clock_ge is
  begin
    process(CLCK)
      variable I : integer := 0;
      begin
        CLCK <= not CLCK after 5 ns;
        I := I + 1;
        assert I <= 80
          report "job done"
          severity Error;
      end process;
    end clk_ctl;
```

----- Serial load parallel shift register -----

```
entity LOAD is
  port (AI : in bit_vector(11 downto 0); B0,B1,B2,B3,B4,B5,B6,B7 : out bit_vector(11
    downto 0);CLK : in bit);
end LOAD;
architecture BEH of LOAD is
  type shift is array(0 to 7)of bit_vector(11 downto 0);
  begin
    process
      variable A : shift;
      variable I,count : integer := 0;
    begin
      wait until CLK'event and CLK = '1';
      for count in 0 to 7 loop
        wait until CLK'event and CLK = '1';
        for I in 0 to 6 loop
          A(I) := A(I+1);
        end loop;
        A(7) := AI;
        if (count = 7) and (CLK'event and CLK='1') then
          B0 <= A(7);
          B1 <= A(6);
```

```

        B2 <= A(5);
        B3 <= A(4);
        B4 <= A(3);
        B5 <= A(2);
        B6 <= A(1);
        B7 <= A(0);
    end if;
end loop;
wait on AI,CLK;
end process;
end BEH;
----- Twice faster clock generator -----
entity clock is
    port(CLK :inout bit := '1');
end clock;
architecture beh of clock is
begin
    process(CLK)
        variable I : integer := 0;
    begin
        CLK <= not CLK after 2.5 ns;
        I := I + 1;
        assert I <= 160
            report "job done"
            severity Error;
    end process;
end beh;
----- Delay gate -----
entity delay10 is
    port(a : bit;b : out bit;CLK : bit);
end delay10;
architecture beh of delay10 is
begin
    process
        variable x : bit;
    begin
        wait until CLK'event and CLK = '1';
        x := a;
        b <= x;
        wait on CLK,a;
    end process;
end beh;
----- Parallel shift out 2-bit register -----

```

```

entity shift is
  port(bi0,bi1,bi2,bi3,bi4,bi5,bi6,bi7 : in bit_vector(11 downto 0);
        bo0,bo1,bo2,bo3,bo4,bo5,bo6,bo7 : out bit_vector(1 downto 0);
        CLK : in bit);
end shift;
architecture beh of shift is
begin
  process
    variable I : integer := 0;
  begin
    wait for 90 ns;
    for r in 0 to 5 loop
      wait until CLK'event and CLK = '1';
      bo0(0) <= bi0(I);
      bo0(1) <= bi0(I+1);
      bo1(0) <= bi1(I);
      bo1(1) <= bi1(I+1);
      bo2(0) <= bi2(I);
      bo2(1) <= bi2(I+1);
      bo3(0) <= bi3(I);
      bo3(1) <= bi3(I+1);
      bo4(0) <= bi4(I);
      bo4(1) <= bi4(I+1);
      bo5(0) <= bi5(I);
      bo5(1) <= bi5(I+1);
      bo6(0) <= bi6(I);
      bo6(1) <= bi6(I+1);
      bo7(0) <= bi7(I);
      bo7(1) <= bi7(I+1);
      I := I + 2;
    end loop;

    I := 0;
    wait on CLK,bi0,bi1,bi2,bi3,bi4,bi5,bi6,bi7;
  end process;
end beh;
----- 2-bit adder/subtractor -----
entity adsu is
  port(a0,a1,a2,a3,a4,a5,a6,a7 : bit_vector(1 downto 0);
        b0,b1,b2,b3,b4,b5,b6,b7 : out bit_vector(1 downto 0);
        CLK,cr,st : bit);
end adsu;

```

architecture beh of adsu is

begin

process

variable c1,c2,c3,c4,c5,c6,c7,c8 : bit_vector(4 downto 0);

variable d1,d2,d3,d4,d5,d6,d7,d8 : bit_vector(2 downto 0);

variable e1,e2,e3,e4,e5,e6,e7,e8 : bit;

begin

wait until CLK'event and CLK = '1';

if cr'event then

e1 := cr; e2 := cr; e3 := cr; e4 := cr;

end if;

if st'event then

e5 := st; e6 := st; e7 := st; e8 := st;

end if;

c1(0) := e1;

c1(1) := a0(0);

c1(2) := a0(1);

c1(3) := a7(0);

c1(4) := a7(1);

d1(0) := (c1(1) and (not c1(3)) and (not c1(0)))

or (not c1(1) and c1(3) and (not c1(0)))

or (not c1(1) and (not c1(3)) and c1(0))

or (c1(1) and c1(3) and c1(0));

d1(1) := (not c1(2) and not c1(1) and c1(4) and not c1(0))

or (not c1(2) and c1(4) and not c1(3) and not c1(0))

or (c1(2) and not c1(4) and not c1(3) and not c1(0))

or (c1(2) and not c1(1) and not c1(4) and not c1(0))

or (not c1(2) and c1(1) and not c1(4) and c1(3))

or (c1(2) and c1(1) and c1(3) and c1(4))

or (not c1(1) and not c1(2) and c1(4) and not c1(3))

or (not c1(1) and c1(2) and not c1(3) and not c1(4))

or (c1(1) and not c1(2) and not c1(4) and c1(0))

or (not c1(2) and not c1(4) and c1(3) and c1(0))

or (c1(2) and c1(3) and c1(4) and c1(0))

or (c1(2) and c1(1) and c1(4) and c1(0));

d1(2) := (c1(1) and c1(2) and c1(3))

or (c1(1) and c1(3) and c1(4))

or (c1(1) and c1(2) and c1(0))

or (c1(2) and c1(3) and c1(0))

or (c1(3) and c1(4) and c1(0))

or (c1(2) and c1(4))

or (c1(1) and c1(4) and c1(0));

b0(0) <= d1(0);
b0(1) <= d1(1);
e1 := d1(2);

c2(0) := e2;
c2(1) := a1(0);
c2(2) := a1(1);
c2(3) := a6(0);
c2(4) := a6(1);
d2(0) := (c2(1) and (not c2(3)) and (not c2(0)))
 or (not c2(1) and c2(3) and (not c2(0)))
 or (not c2(1) and (not c2(3)) and c2(0))
 or (c2(1) and c2(3) and c2(0));
d2(1) := (not c2(2) and not c2(1) and c2(4) and not c2(0))
 or (not c2(2) and c2(4) and not c2(3) and not c2(0))
 or (c2(2) and not c2(4) and not c2(3) and not c2(0))
 or (c2(2) and not c2(1) and not c2(4) and not c2(0))
 or (not c2(2) and c2(1) and not c2(4) and c2(3))
 or (c2(2) and c2(1) and c2(3) and c2(4))
 or (not c2(1) and not c2(2) and c2(4) and not c2(3))
 or (not c2(1) and c2(2) and not c2(3) and not c2(4))
 or (c2(1) and not c2(2) and not c2(4) and c2(0))
 or (not c2(2) and not c2(4) and c2(3) and c2(0))
 or (c2(2) and c2(3) and c2(4) and c2(0))
 or (c2(2) and c2(1) and c2(4) and c2(0));
d2(2) := (c2(1) and c2(2) and c2(3))
 or (c2(1) and c2(3) and c2(4))
 or (c2(1) and c2(2) and c2(0))
 or (c2(2) and c2(3) and c2(0))
 or (c2(3) and c2(4) and c2(0))
 or (c2(2) and c2(4))
 or (c2(1) and c2(4) and c2(0));
b1(0) <= d2(0);
b1(1) <= d2(1);
e2 := d2(2);

c3(0) := e3;
c3(1) := a2(0);
c3(2) := a2(1);
c3(3) := a5(0);
c3(4) := a5(1);
d3(0) := (c3(1) and (not c3(3)) and (not c3(0)))
 or (not c3(1) and c3(3) and (not c3(0)))

```

    or (not c3(1) and (not c3(3)) and c3(0))
    or (c3(1) and c3(3) and c3(0));
d3(1) := (not c3(2) and not c3(1) and c3(4) and not c3(0))
    or (not c3(2) and c3(4) and not c3(3) and not c3(0))
    or (c3(2) and not c3(4) and not c3(3) and not c3(0))
    or (c3(2) and not c3(1) and not c3(4) and not c3(0))
    or (not c3(2) and c3(1) and not c3(4) and c3(3))
    or (c3(2) and c3(1) and c3(3) and c3(4))
    or (not c3(1) and not c3(2) and c3(4) and not c3(3))
    or (not c3(1) and c3(2) and not c3(3) and not c3(4))
    or (c3(1) and not c3(2) and not c3(4) and c3(0))
    or (not c3(2) and not c3(4) and c3(3) and c3(0))
    or (c3(2) and c3(3) and c3(4) and c3(0))
    or (c3(2) and c3(1) and c3(4) and c3(0));
d3(2) := (c3(1) and c3(2) and c3(3))
    or (c3(1) and c3(3) and c3(4))
    or (c3(1) and c3(2) and c3(0))
    or (c3(2) and c3(3) and c3(0))
    or (c3(3) and c3(4) and c3(0))
    or (c3(2) and c3(4))
    or (c3(1) and c3(4) and c3(0));
b2(0) <= d3(0);
b2(1) <= d3(1);
e3 := d3(2);

```

```

c4(0) := e4;
c4(1) := a3(0);
c4(2) := a3(1);
c4(3) := a4(0);
c4(4) := a4(1);
d4(0) := (c4(1) and (not c4(3)) and (not c4(0)))
    or (not c4(1) and c4(3) and (not c4(0)))
    or (not c4(1) and (not c4(3)) and c4(0))
    or (c4(1) and c4(3) and c4(0));
d4(1) := (not c4(2) and not c4(1) and c4(4) and not c4(0))
    or (not c4(2) and c4(4) and not c4(3) and not c4(0))
    or (c4(2) and not c4(4) and not c4(3) and not c4(0))
    or (c4(2) and not c4(1) and not c4(4) and not c4(0))
    or (not c4(2) and c4(1) and not c4(4) and c4(3))
    or (c4(2) and c4(1) and c4(3) and c4(4))
    or (not c4(1) and not c4(2) and c4(4) and not c4(3))
    or (not c4(1) and c4(2) and not c4(3) and not c4(4))
    or (c4(1) and not c4(2) and not c4(4) and c4(0))

```

```

    or (not c4(2) and not c4(4) and c4(3) and c4(0))
    or (c4(2) and c4(3) and c4(4) and c4(0))
    or (c4(2) and c4(1) and c4(4) and c4(0));
d4(2) := (c4(1) and c4(2) and c4(3))
    or (c4(1) and c4(3) and c4(4))
    or (c4(1) and c4(2) and c4(0))
    or (c4(2) and c4(3) and c4(0))
    or (c4(3) and c4(4) and c4(0))
    or (c4(2) and c4(4))
    or (c4(1) and c4(4) and c4(0));
b3(0) <= d4(0);
b3(1) <= d4(1);
e4 := d4(2);

```

```

c5(0) := e5;
c5(1) := a3(0);
c5(2) := a3(1);
c5(3) := not a4(0);
c5(4) := not a4(1);
d5(0) := (c5(1) and (not c5(3)) and (not c5(0)))
    or (not c5(1) and c5(3) and (not c5(0)))
    or (not c5(1) and (not c5(3)) and c5(0))
    or (c5(1) and c5(3) and c5(0));
d5(1) := (not c5(2) and not c5(1) and c5(4) and not c5(0))
    or (not c5(2) and c5(4) and not c5(3) and not c5(0))
    or (c5(2) and not c5(4) and not c5(3) and not c5(0))
    or (c5(2) and not c5(1) and not c5(4) and not c5(0))
    or (not c5(2) and c5(1) and not c5(4) and c5(3))
    or (c5(2) and c5(1) and c5(3) and c5(4))
    or (not c5(1) and not c5(2) and c5(4) and not c5(3))
    or (not c5(1) and c5(2) and not c5(3) and not c5(4))
    or (c5(1) and not c5(2) and not c5(4) and c5(0))
    or (not c5(2) and not c5(4) and c5(3) and c5(0))
    or (c5(2) and c5(3) and c5(4) and c5(0))
    or (c5(2) and c5(1) and c5(4) and c5(0));
d5(2) := (c5(1) and c5(2) and c5(3))
    or (c5(1) and c5(3) and c5(4))
    or (c5(1) and c5(2) and c5(0))
    or (c5(2) and c5(3) and c5(0))
    or (c5(3) and c5(4) and c5(0))
    or (c5(2) and c5(4))
    or (c5(1) and c5(4) and c5(0));
b4(0) <= d5(0);

```


b4(1) <= d5(1);
e5 := d5(2);

c6(0) := e6;
c6(1) := a2(0);
c6(2) := a2(1);
c6(3) := not a5(0);
c6(4) := not a5(1);
d6(0) := (c6(1) and (not c6(3)) and (not c6(0)))
or (not c6(1) and c6(3) and (not c6(0)))
or (not c6(1) and (not c6(3)) and c6(0))
or (c6(1) and c6(3) and c6(0));
d6(1) := (not c6(2) and not c6(1) and c6(4) and not c6(0))
or (not c6(2) and c6(4) and not c6(3) and not c6(0))
or (c6(2) and not c6(4) and not c6(3) and not c6(0))
or (c6(2) and not c6(1) and not c6(4) and not c6(0))
or (not c6(2) and c6(1) and not c6(4) and c6(3))
or (c6(2) and c6(1) and c6(3) and c6(4))
or (not c6(1) and not c6(2) and c6(4) and not c6(3))
or (not c6(1) and c6(2) and not c6(3) and not c6(4))
or (c6(1) and not c6(2) and not c6(4) and c6(0))
or (not c6(2) and not c6(4) and c6(3) and c6(0))
or (c6(2) and c6(3) and c6(4) and c6(0))
or (c6(2) and c6(1) and c6(4) and c6(0));
d6(2) := (c6(1) and c6(2) and c6(3))
or (c6(1) and c6(3) and c6(4))
or (c6(1) and c6(2) and c6(0))
or (c6(2) and c6(3) and c6(0))
or (c6(3) and c6(4) and c6(0))
or (c6(2) and c6(4))
or (c6(1) and c6(4) and c6(0));
b5(0) <= d6(0);
b5(1) <= d6(1);
e6 := d6(2);

c7(0) := e7;
c7(1) := a1(0);
c7(2) := a1(1);
c7(3) := not a6(0);
c7(4) := not a6(1);
d7(0) := (c7(1) and (not c7(3)) and (not c7(0)))
or (not c7(1) and c7(3) and (not c7(0)))
or (not c7(1) and (not c7(3)) and c7(0))

```

    or (c7(1) and c7(3) and c7(0));
d7(1) := (not c7(2) and not c7(1) and c7(4) and not c7(0))
    or (not c7(2) and c7(4) and not c7(3) and not c7(0))
    or (c7(2) and not c7(4) and not c7(3) and not c7(0))
    or (c7(2) and not c7(1) and not c7(4) and not c7(0))
    or (not c7(2) and c7(1) and not c7(4) and c7(3))
    or (c7(2) and c7(1) and c7(3) and c7(4))
    or (not c7(1) and not c7(2) and c7(4) and not c7(3))
    or (not c7(1) and c7(2) and not c7(3) and not c7(4))
    or (c7(1) and not c7(2) and not c7(4) and c7(0))
    or (not c7(2) and not c7(4) and c7(3) and c7(0))
    or (c7(2) and c7(3) and c7(4) and c7(0))
    or (c7(2) and c7(1) and c7(4) and c7(0));
d7(2) := (c7(1) and c7(2) and c7(3))
    or (c7(1) and c7(3) and c7(4))
    or (c7(1) and c7(2) and c7(0))
    or (c7(2) and c7(3) and c7(0))
    or (c7(3) and c7(4) and c7(0))
    or (c7(2) and c7(4))
    or (c7(1) and c7(4) and c7(0));
b6(0) <= d7(0);
b6(1) <= d7(1);
e7 := d7(2);

```

```

c8(0) := e8;
c8(1) := a0(0);
c8(2) := a0(1);
c8(3) := not a7(0);
c8(4) := not a7(1);
d8(0) := (c8(1) and (not c8(3)) and (not c8(0)))
    or (not c8(1) and c8(3) and (not c8(0)))
    or (not c8(1) and (not c8(3)) and c8(0))
    or (c8(1) and c8(3) and c8(0));
d8(1) := (not c8(2) and not c8(1) and c8(4) and not c8(0))
    or (not c8(2) and c8(4) and not c8(3) and not c8(0))
    or (c8(2) and not c8(4) and not c8(3) and not c8(0))
    or (c8(2) and not c8(1) and not c8(4) and not c8(0))
    or (not c8(2) and c8(1) and not c8(4) and c8(3))
    or (c8(2) and c8(1) and c8(3) and c8(4))
    or (not c8(1) and not c8(2) and c8(4) and not c8(3))
    or (not c8(1) and c8(2) and not c8(3) and not c8(4))
    or (c8(1) and not c8(2) and not c8(4) and c8(0))
    or (not c8(2) and not c8(4) and c8(3) and c8(0))

```

```

        or (c8(2) and c8(3) and c8(4) and c8(0))
        or (c8(2) and c8(1) and c8(4) and c8(0));
d8(2) := (c8(1) and c8(2) and c8(3))
        or (c8(1) and c8(3) and c8(4))
        or (c8(1) and c8(2) and c8(0))
        or (c8(2) and c8(3) and c8(0))
        or (c8(3) and c8(4) and c8(0))
        or (c8(2) and c8(4))
        or (c8(1) and c8(4) and c8(0));
b7(0) <= d8(0);
b7(1) <= d8(1);
e8 := d8(2);
wait on a0,a1,a2,a3,a4,a5,a6,a7,CLK,cr,st;
end process;
end beh;

----- Register -----
entity reg is
    port(a0,a1,a2,a3,a4,a5,a6,a7 : bit_vector(1 downto 0);
          b0,b1,b2,b3,b4,b5,b6,b7 : out bit_vector(1 downto 0);
          CLK : bit);
end reg;
architecture beh of reg is
begin
    process
        variable d0,d1,d2,d3,d4,d5,d6,d7 : bit_vector(1 downto 0);
    begin
        d0 := a0;
        d1 := a1;
        d2 := a2;
        d3 := a3;
        d4 := a4;
        d5 := a5;
        d6 := a6;
        d7 := a7;
        wait until CLK'event and CLK = '1';
        b0 <= d0;
        b1 <= d1;
        b2 <= d2;
        b3 <= d3;
        b4 <= d4;
        b5 <= d5;
        b6 <= d6;
        b7 <= d7;
    end process;
end beh;

```

```

        wait on CLK;
    end process;
end beh;
----- ROM -----
entity rom is
    port(e0,e1,e2,e3,e4,e5,e6,e7 : bit_vector(1 downto 0);
        b10,b11,b20,b21,b30,b31,b40,b41,b50,b51,b60,b61,b70,b71,b80,b81:
        out bit_vector(15 downto 0);
        CLK : bit);
end rom;
architecture beh of rom is
begin
    process
        variable a10,a11,a20,a21,a30,a31,a40,a41,a50,a51,a60,a61,a70,a71,
            a80,a81 : bit_vector(3 downto 0);
    begin
        wait until CLK'event and CLK = '1';
        a10(3) := e0(0); a10(2) := e1(0); a10(1) := e2(0); a10(0) := e3(0);
        a11(3) := e0(1); a11(2) := e1(1); a11(1) := e2(1); a11(0) := e3(1);
        a20(3) := e7(0); a20(2) := e6(0); a20(1) := e5(0); a20(0) := e4(0);
        a21(3) := e7(1); a21(2) := e6(1); a21(1) := e5(1); a21(0) := e4(1);
        a30(3) := e0(0); a30(2) := e1(0); a30(1) := e2(0); a30(0) := e3(0);
        a31(3) := e0(1); a31(2) := e1(1); a31(1) := e2(1); a31(0) := e3(1);
        a40(3) := e7(0); a40(2) := e6(0); a40(1) := e5(0); a40(0) := e4(0);
        a41(3) := e7(1); a41(2) := e6(1); a41(1) := e5(1); a41(0) := e4(1);
        a50(3) := e0(0); a50(2) := e1(0); a50(1) := e2(0); a50(0) := e3(0);
        a51(3) := e0(1); a51(2) := e1(1); a51(1) := e2(1); a51(0) := e3(1);
        a60(3) := e7(0); a60(2) := e6(0); a60(1) := e5(0); a60(0) := e4(0);
        a61(3) := e7(1); a61(2) := e6(1); a61(1) := e5(1); a61(0) := e4(1);
        a70(3) := e0(0); a70(2) := e1(0); a70(1) := e2(0); a70(0) := e3(0);
        a71(3) := e0(1); a71(2) := e1(1); a71(1) := e2(1); a71(0) := e3(1);
        a80(3) := e7(0); a80(2) := e6(0); a80(1) := e5(0); a80(0) := e4(0);
        a81(3) := e7(1); a81(2) := e6(1); a81(1) := e5(1); a81(0) := e4(1);

        case a10 is
            when "0000" => b10 <= "0000000000000000";
            when "0001" => b10 <= "0001011010100000";
            when "0010" => b10 <= "0001011010100000";
            when "0011" => b10 <= "0010110101000001";
            when "0100" => b10 <= "0001011010100000";
            when "0101" => b10 <= "0010110101000001";
            when "0110" => b10 <= "0010110101000001";
            when "0111" => b10 <= "0100001111100001";
        end case;
    end process;
end beh;

```

```

when "1000" => b10 <= "0001011010100000";
when "1001" => b10 <= "0010110101000001";
when "1010" => b10 <= "0010110101000001";
when "1011" => b10 <= "0100001111100001";
when "1100" => b10 <= "0010110101000001";
when "1101" => b10 <= "0100001111100001";
when "1110" => b10 <= "0100001111100001";
when "1111" => b10 <= "0101101010000010";
end case;

```

```

case a11 is
when "0000" => b11 <= "0000000000000000";
when "0001" => b11 <= "0001011010100000";
when "0010" => b11 <= "0001011010100000";
when "0011" => b11 <= "0010110101000001";
when "0100" => b11 <= "0001011010100000";
when "0101" => b11 <= "0010110101000001";
when "0110" => b11 <= "0010110101000001";
when "0111" => b11 <= "0100001111100001";
when "1000" => b11 <= "0001011010100000";
when "1001" => b11 <= "0010110101000001";
when "1010" => b11 <= "0010110101000001";
when "1011" => b11 <= "0100001111100001";
when "1100" => b11 <= "0010110101000001";
when "1101" => b11 <= "0100001111100001";
when "1110" => b11 <= "0100001111100001";
when "1111" => b11 <= "0101101010000010";
end case;

```

```

case a20 is
when "0000" => b20 <= "0000000000000000";
when "0001" => b20 <= "00000011000111110";
when "0010" => b20 <= "0001000111000111";
when "0011" => b20 <= "0001100000000101";
when "0100" => b20 <= "0001101010011011";
when "0101" => b20 <= "0010000011011001";
when "0110" => b20 <= "0010110001100010";
when "0111" => b20 <= "0011001010100000";
when "1000" => b20 <= "0001111101100010";
when "1001" => b20 <= "0010010110100000";
when "1010" => b20 <= "0011000100101001";
when "1011" => b20 <= "0011011101101000";
when "1100" => b20 <= "0011100111111101";

```

```

when "1101" => b20 <= "0100000000111100";
when "1110" => b20 <= "0100101111000101";
when "1111" => b20 <= "0101001000000011";
end case;

```

```

case a21 is
when "0000" => b21 <= "0000000000000000";
when "0001" => b21 <= "0000011000111110";
when "0010" => b21 <= "0001000111000111";
when "0011" => b21 <= "000110000000101";
when "0100" => b21 <= "0001101010011011";
when "0101" => b21 <= "0010000011011001";
when "0110" => b21 <= "0010110001100010";
when "0111" => b21 <= "0011001010100000";
when "1000" => b21 <= "0001111101100010";
when "1001" => b21 <= "0010010110100000";
when "1010" => b21 <= "0011000100101001";
when "1011" => b21 <= "0011011101101000";
when "1100" => b21 <= "0011100111111101";
when "1101" => b21 <= "010000000111100";
when "1110" => b21 <= "0100101111000101";
when "1111" => b21 <= "0101001000000011";
end case;

```

```

case a30 is
when "0000" => b30 <= "0000000000000000";
when "0001" => b30 <= "1110001001110000";
when "0010" => b30 <= "1111001111000010";
when "0011" => b30 <= "1101011000110001";
when "0100" => b30 <= "0000110000111110";
when "0101" => b30 <= "1110111010101111";
when "0110" => b30 <= "0000000000000000";
when "0111" => b30 <= "1110001001110000";
when "1000" => b30 <= "0001110110010000";
when "1001" => b30 <= "0000000000000000";
when "1010" => b30 <= "0001000101010001";
when "1011" => b30 <= "1111001111000010";
when "1100" => b30 <= "0010100111001111";
when "1101" => b30 <= "0000110000111110";
when "1110" => b30 <= "0001110110010000";
when "1111" => b30 <= "0000000000000000";
end case;

```

case a31 is

```
when "0000" => b31 <= "0000000000000000";
when "0001" => b31 <= "1110001001110000";
when "0010" => b31 <= "1111001111000010";
when "0011" => b31 <= "1101011000110001";
when "0100" => b31 <= "0000110000111110";
when "0101" => b31 <= "1110111010101111";
when "0110" => b31 <= "0000000000000000";
when "0111" => b31 <= "1110001001110000";
when "1000" => b31 <= "0001110110010000";
when "1001" => b31 <= "0000000000000000";
when "1010" => b31 <= "0001000101010001";
when "1011" => b31 <= "1111001111000010";
when "1100" => b31 <= "0010100111001111";
when "1101" => b31 <= "0000110000111110";
when "1110" => b31 <= "0001110110010000";
when "1111" => b31 <= "0000000000000000";
end case;
```

case a40 is

```
when "0000" => b40 <= "0000000000000000";
when "0001" => b40 <= "1110111000111001";
when "0010" => b40 <= "1110000010011110";
when "0011" => b40 <= "1100111011010111";
when "0100" => b40 <= "1111100111000010";
when "0101" => b40 <= "1110011111111011";
when "0110" => b40 <= "1101101001100000";
when "0111" => b40 <= "1100100010011000";
when "1000" => b40 <= "0001101010011011";
when "1001" => b40 <= "0000100011010100";
when "1010" => b40 <= "1111101100111001";
when "1011" => b40 <= "1110100101110010";
when "1100" => b40 <= "0001010001011101";
when "1101" => b40 <= "0000001010010101";
when "1110" => b40 <= "1111010011111011";
when "1111" => b40 <= "1110001100110100";
end case;
```

case a41 is

```
when "0000" => b41 <= "0000000000000000";
when "0001" => b41 <= "1110111000111001";
when "0010" => b41 <= "1110000010011110";
```

```

when "0011" => b41 <= "1100111011010111";
when "0100" => b41 <= "1111100111000010";
when "0101" => b41 <= "1110011111111011";
when "0110" => b41 <= "1101101001100000";
when "0111" => b41 <= "1100100010011000";
when "1000" => b41 <= "0001101010011011";
when "1001" => b41 <= "0000100011010101";
when "1010" => b41 <= "1111101100111001";
when "1011" => b41 <= "1110100101110010";
when "1100" => b41 <= "0001010001011101";
when "1101" => b41 <= "0000001010010101";
when "1110" => b41 <= "1111010011111011";
when "1111" => b41 <= "1110001100110100";
end case;

```

```

case a50 is
when "0000" => b50 <= "0000000000000000";
when "0001" => b50 <= "0001011010100000";
when "0010" => b50 <= "1110100101100000";
when "0011" => b50 <= "0000000000000000";
when "0100" => b50 <= "1110100101100000";
when "0101" => b50 <= "0000000000000000";
when "0110" => b50 <= "1101001010111111";
when "0111" => b50 <= "1110100101100000";
when "1000" => b50 <= "0001011010100000";
when "1001" => b50 <= "0010110101000001";
when "1010" => b50 <= "0000000000000000";
When "1011" => b50 <= "0001011010100000";
When "1100" => b50 <= "0000000000000000";
When "1101" => b50 <= "0001011010100000";
When "1110" => b50 <= "1110100101100000";
When "1111" => b50 <= "0000000000000000";
end case;

```

```

case a51 is
when "0000" => b51 <= "0000000000000000";
when "0001" => b51 <= "0001011010100000";
when "0010" => b51 <= "1110100101100000";
when "0011" => b51 <= "0000000000000000";
when "0100" => b51 <= "1110100101100000";
when "0101" => b51 <= "0000000000000000";
when "0110" => b51 <= "1101001010111111";
when "0111" => b51 <= "1110100101100000";

```



```

when "1000" => b51 <= "0001011010100000";
when "1001" => b51 <= "0010110101000001";
when "1010" => b51 <= "0000000000000000";
When "1011" => b51 <= "0001011010100000";
When "1100" => b51 <= "0000000000000000";
When "1101" => b51 <= "0001011010100000";
When "1110" => b51 <= "1110100101100000";
When "1111" => b51 <= "0000000000000000";
end case;

```

```

case a60 is
when "0000" => b60 <= "0000000000000000";
when "0001" => b60 <= "0001101010011011";
when "0010" => b60 <= "0000011000111110";
when "0011" => b60 <= "0010000011011001";
when "0100" => b60 <= "1110000010011110";
when "0101" => b60 <= "1111101100111001";
when "0110" => b60 <= "1110011011011100";
when "0111" => b60 <= "0000000101110110";
when "1000" => b60 <= "0001000111000111";
when "1001" => b60 <= "0010110001100010";
when "1010" => b60 <= "0001100000000101";
When "1011" => b60 <= "0011001010100000";
When "1100" => b60 <= "1111001001100101";
When "1101" => b60 <= "0000110100000000";
When "1110" => b60 <= "1111100010100011";
When "1111" => b60 <= "0001001100111110";
end case;

```

```

case a61 is
when "0000" => b61 <= "0000000000000000";
when "0001" => b61 <= "0001101010011011";
when "0010" => b61 <= "0000011000111110";
when "0011" => b61 <= "0010000011011001";
when "0100" => b61 <= "1110000010011110";
when "0101" => b61 <= "1111101100111001";
when "0110" => b61 <= "1110011011011100";
when "0111" => b61 <= "0000000101110110";
when "1000" => b61 <= "0001000111000111";
when "1001" => b61 <= "0010110001100010";
when "1010" => b61 <= "0001100000000101";
When "1011" => b61 <= "0011001010100000";
When "1100" => b61 <= "1111001001100101";

```

```

When "1101" => b61 <= "0000110100000000";
When "1110" => b61 <= "1111100010100011";
When "1111" => b61 <= "0001001100111110";
end case;

```

```

case a70 is
when "0000" => b70 <= "0000000000000000";
when "0001" => b70 <= "1111001111000010";
when "0010" => b70 <= "0001110110010000";
when "0011" => b70 <= "0001000101010001";
when "0100" => b70 <= "1110001001110000";
when "0101" => b70 <= "1101011000110001";
when "0110" => b70 <= "0000000000000000";
when "0111" => b70 <= "1111001111000010";
when "1000" => b70 <= "0000110000111110";
when "1001" => b70 <= "0000000000000000";
when "1010" => b70 <= "0010100111001111";
When "1011" => b70 <= "0001110110010000";
When "1100" => b70 <= "1110111010101111";
When "1101" => b70 <= "1110001001110000";
When "1110" => b70 <= "0000110000111110";
When "1111" => b70 <= "0000000000000000";
end case;

```

```

case a71 is
when "0000" => b71 <= "0000000000000000";
when "0001" => b71 <= "1111001111000010";
when "0010" => b71 <= "0001110110010000";
when "0011" => b71 <= "0001000101010001";
when "0100" => b71 <= "1110001001110000";
when "0101" => b71 <= "1101011000110001";
when "0110" => b71 <= "0000000000000000";
when "0111" => b71 <= "1111001111000010";
when "1000" => b71 <= "0000110000111110";
when "1001" => b71 <= "0000000000000000";
when "1010" => b71 <= "0010100111001111";
When "1011" => b71 <= "0001110110010000";
When "1100" => b71 <= "1110111010101111";
When "1101" => b71 <= "1110001001110000";
When "1110" => b71 <= "0000110000111110";
When "1111" => b71 <= "0000000000000000";
end case;

```

```

case a80 is
when "0000" => b80 <= "0000000000000000";
when "0001" => b80 <= "1110000010011110";
when "0010" => b80 <= "0001101010011011";
when "0011" => b80 <= "1111101100111001";
when "0100" => b80 <= "1110111000111001";
when "0101" => b80 <= "1100111011010111";
when "0110" => b80 <= "0000100011010100";
when "0111" => b80 <= "1110100101110010";
when "1000" => b80 <= "0000011000111110";
when "1001" => b80 <= "1110011011011100";
when "1010" => b80 <= "0010000011011001";
When "1011" => b80 <= "0000000101110110";
When "1100" => b80 <= "1111010001110111";
When "1101" => b80 <= "1101010100010101";
When "1110" => b80 <= "0000111100010010";
When "1111" => b80 <= "1110111110110000";
end case;

```

```

case a81 is
when "0000" => b81 <= "0000000000000000";
when "0001" => b81 <= "1110000010011110";
when "0010" => b81 <= "0001101010011011";
when "0011" => b81 <= "1111101100111001";
when "0100" => b81 <= "1110111000111001";
when "0101" => b81 <= "1100111011010111";
when "0110" => b81 <= "0000100011010100";
when "0111" => b81 <= "1110100101110010";
when "1000" => b81 <= "0000011000111110";
when "1001" => b81 <= "1110011011011100";
when "1010" => b81 <= "0010000011011001";
When "1011" => b81 <= "0000000101110110";
When "1100" => b81 <= "1111010001110111";
When "1101" => b81 <= "1101010100010101";
When "1110" => b81 <= "0000111100010010";
When "1111" => b81 <= "1110111110110000";
end case;

```

```

wait on e0,e1,e2,e3,e4,e5,e6,e7,CLK;

```

```

end process;

```

```

end beh;

```

Shift right 1-bit register

```

entity shi_1 is

```

```

port(f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,f13,f14,f15,f16:

```

```

    bit_vector(15 downto 0);
    b10,b11,b20,b21,b30,b31,b40,b41,b50,b51,b60,b61,b70,b71,b80,b81:
    out bit_vector(15 downto 0);
    CLK : bit);
end shi_1;
architecture beh of shi_1 is
begin
    process
        variable a1,a2,a3,a4,a5,a6,a7,a8 : bit_vector(15 downto 0);
    begin
        wait until CLK'event and CLK = '1';
        if f1(15) = '0' then
            a1(15) := '0';
        else
            a1(15) := '1';
        end if;
        a1(14) := f1(15);
        a1(13) := f1(14);
        a1(12) := f1(13);
        a1(11) := f1(12);
        a1(10) := f1(11);
        a1(9) := f1(10);
        a1(8) := f1(9);
        a1(7) := f1(8);
        a1(6) := f1(7);
        a1(5) := f1(6);
        a1(4) := f1(5);
        a1(3) := f1(4);
        a1(2) := f1(3);
        a1(1) := f1(2);
        a1(0) := f1(1);
        b10 <= a1;
        b11 <= f2;
    end process;
    -----
    if f3(15) = '0' then
        a2(15) := '0';
    else
        a2(15) := '1';
    end if;
    a2(14) := f3(15);
    a2(13) := f3(14);
    a2(12) := f3(13);
    a2(11) := f3(12);

```

```
a2(10) := f3(11);
a2(9) := f3(10);
a2(8) := f3(9);
a2(7) := f3(8);
a2(6) := f3(7);
a2(5) := f3(6);
a2(4) := f3(5);
a2(3) := f3(4);
a2(2) := f3(3);
a2(1) := f3(2);
a2(0) := f3(1);
b20 <= a2;
b21 <= f4;
```

```
if f5(15) = '0' then
  a3(15) := '0';
else
  a3(15) := '1';
end if;
a3(14) := f5(15);
a3(13) := f5(14);
a3(12) := f5(13);
a3(11) := f5(12);
a3(10) := f5(11);
a3(9) := f5(10);
a3(8) := f5(9);
a3(7) := f5(8);
a3(6) := f5(7);
a3(5) := f5(6);
a3(4) := f5(5);
a3(3) := f5(4);
a3(2) := f5(3);
a3(1) := f5(2);
a3(0) := f5(1);
b30 <= a3;
b31 <= f6;
```

```
if f7(15) = '0' then
  a4(15) := '0';
else
  a4(15) := '1';
end if;
a4(14) := f7(15);
```

```
a4(13) := f7(14);
a4(12) := f7(13);
a4(11) := f7(12);
a4(10) := f7(11);
a4(9) := f7(10);
a4(8) := f7(9);
a4(7) := f7(8);
a4(6) := f7(7);
a4(5) := f7(6);
a4(4) := f7(5);
a4(3) := f7(4);
a4(2) := f7(3);
a4(1) := f7(2);
a4(0) := f7(1);
b40 <= a4;
b41 <= f8;
```

```
if f9(15) = '0' then
  a5(15) := '0';
else
  a5(15) := '1';
end if;
a5(14) := f9(15);
a5(13) := f9(14);
a5(12) := f9(13);
a5(11) := f9(12);
a5(10) := f9(11);
a5(9) := f9(10);
a5(8) := f9(9);
a5(7) := f9(8);
a5(6) := f9(7);
a5(5) := f9(6);
a5(4) := f9(5);
a5(3) := f9(4);
a5(2) := f9(3);
a5(1) := f9(2);
a5(0) := f9(1);
b50 <= a5;
b51 <= f10;
```

```
if f11(15) = '0' then
  a6(15) := '0';
else
```

```

a6(15) := '1';
end if;
a6(14) := f11(15);
a6(13) := f11(14);
a6(12) := f11(13);
a6(11) := f11(12);
a6(10) := f11(11);
a6(9) := f11(10);
a6(8) := f11(9);
a6(7) := f11(8);
a6(6) := f11(7);
a6(5) := f11(6);
a6(4) := f11(5);
a6(3) := f11(4);
a6(2) := f11(3);
a6(1) := f11(2);
a6(0) := f11(1);
b60 <= a6;
b61 <= f12;

```

```

if f13(15) = '0' then
a7(15) := '0';
else
a7(15) := '1';
end if;
a7(14) := f13(15);
a7(13) := f13(14);
a7(12) := f13(13);
a7(11) := f13(12);
a7(10) := f13(11);
a7(9) := f13(10);
a7(8) := f13(9);
a7(7) := f13(8);
a7(6) := f13(7);
a7(5) := f13(6);
a7(4) := f13(5);
a7(3) := f13(4);
a7(2) := f13(3);
a7(1) := f13(2);
a7(0) := f13(1);
b70 <= a7;
b71 <= f14;

```

```

    if f15(15) = '0' then
        a8(15) := '0';
    else
        a8(15) := '1';
    end if;
    a8(14) := f15(15);
    a8(13) := f15(14);
    a8(12) := f15(13);
    a8(11) := f15(12);
    a8(10) := f15(11);
    a8(9) := f15(10);
    a8(8) := f15(9);
    a8(7) := f15(8);
    a8(6) := f15(7);
    a8(5) := f15(6);
    a8(4) := f15(5);
    a8(3) := f15(4);
    a8(2) := f15(3);
    a8(1) := f15(2);
    a8(0) := f15(1);
    b80 <= a8;
    b81 <= f16;
    wait on f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,f13,f14,f15,f16,CLK;
end process;
end beh;
----- Package 1 -----
package pack1 is
    procedure bi_to_in --change 16 bits(1 sign,1 integer and 14 fraction into real)
        (variable x : bit_vector(15 downto 0);
         variable y : out integer);
    procedure in_to_bi --change real into binary(1 sign,1 integer,14 fractions).
        (variable m : in integer;
         variable n : out bit_vector(15 downto 0));
end pack1;
package body pack1 is
    procedure bi_to_in
        (variable x : bit_vector(15 downto 0);
         variable y : out integer) is
        variable sum : integer := 0;
        variable p : bit_vector(15 downto 0);
    begin
        p := x;
        if p(15) = '1' then

```



```

    for i in 0 to 14 loop
        if p(i) = '1' then
            for i in 0 to 13 loop
                p(i+1) := not p(i+1);
            end loop; exit;
        end if;
    end loop;
    for k in 0 to 14 loop
        if p(k) = '1' then
            sum := sum + 2**k;
        end if;
    end loop;
    y := -sum;
else
    for l in 0 to 14 loop
        if p(l) = '1' then
            sum := sum + 2**l;
        end if;
    end loop;
    y := sum;
end if;
end bi_to_in;

```

```

procedure in_to_bi
    (variable m : in integer;
     variable n : out bit_vector(15 downto 0)) is
    variable temp_a : integer := 0;
    variable temp_b : integer := 0;
    variable w : bit_vector(15 downto 0);
begin
    if m < 0 then
        temp_a := -m;
    else
        temp_a := m;
    end if;
    for i in 14 downto 0 loop
        temp_b := temp_a/(2**i);
        temp_a := temp_a rem (2**i);
        if (temp_b = 1) then
            w(i) := '1';
        else
            w(i) := '0';
        end if;
    end loop;
end in_to_bi;

```

```

        end loop;
    if m > 0 then
        w(15) := '0';
    else
        w(15) := '1';
        for k in 0 to 14 loop
            if w(k) = '1' then
                for k in 0 to 13 loop
                    w(k+1) := not w(k+1);
                end loop; exit;
            end if;
        end loop;
    end if;
end if;
-----
-- prevent negative zero occurs.
    if w(14)='0' and w(13)='0' and w(12)='0' and w(11)='0' and
        w(10)='0' and
        w(9)='0' and w(8)='0' and w(7)='0' and w(6)='0' and w(5)='0' and
        w(4)='0' and w(3)='0' and w(2)='0' and w(1)='0' and w(0)='0' then
        w(15) := '0';
    end if;
    n := w;
end in_to_bi;
end pack1;
----- 16-bit adder_g -----
use work.pack1.all;
entity add_g is
    port(a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13,a14,a15,a16:
        bit_vector(15 downto 0);
        b1,b2,b3,b4,b5, 6,b7,b8 : out bit_vector(15 downto 0);
        CLK,as : bit);
end add_g;
architecture beh of add_g is
    begin
        process
            variable x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,x13,x14,x15,x16,
                n1,n2,n3,n4,n5,n6,n7,n8 : bit_vector(15 downto 0);
            variable y1,y2,y3,y4,y5,y6,y7,y8,y9,y10,y11,y12,y13,y14,y15,y16,
                m1,m2,m3,m4,m5,m6,m7,m8 : integer := 0;
        begin
            wait until CLK'event and CLK = '1';
            x1 := a1; x2 := a2; x3 := a3; x4 := a4;
            x5 := a5; x6 := a6; x7 := a7; x8 := a8;

```

```

x9 := a9; x10 := a10; x11 := a11; x12 := a12;
x13 := a13; x14 := a14; x15 := a15; x16 := a16;
bi_to_in(x1,y1);bi_to_in(x2,y2);bi_to_in(x3,y3);bi_to_in(x4,y4);
bi_to_in(x5,y5);bi_to_in(x6,y6);bi_to_in(x7,y7);bi_to_in(x8,y8);
bi_to_in(x9,y9);bi_to_in(x10,y10);bi_to_in(x11,y11);
bi_to_in(x12,y12);
bi_to_in(x13,y13);bi_to_in(x14,y14);bi_to_in(x15,y15);
bi_to_in(x16,y16);
if as = '0' then
m1 := y1 + y2; m2 := y3 + y4; m3 := y5 + y6; m4 := y7 + y8;
m5 := y9 + y10; m6 := y11 + y12; m7 := y13 + y14; m8 := y15 + y16;
else
m1 := y1 - y2; m2 := y3 - y4; m3 := y5 - y6; m4 := y7 - y8;
m5 := y9 - y10; m6 := y11 - y12; m7 := y13 - y14; m8 := y15 - y16;
end if;
in_to_bi(m1,n1); in_to_bi(m2,n2); in_to_bi(m3,n3); in_to_bi(m4,n4);
in_to_bi(m5,n5); in_to_bi(m6,n6); in_to_bi(m7,n7); in_to_bi(m8,n8);
b1 <= n1; b2 <= n2; b3 <= n3; b4 <= n4;
b5 <= n5; b6 <= n6; b7 <= n7; b8 <= n8;
wait on a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13,a14,a15,a16,CLK;
end process;
end beh;
----- Register_h -----
entity reg_h is
port(a0,a1,a2,a3,a4,a5,a6,a7 : bit_vector(15 downto 0);
      b0,b1,b2,b3,b4,b5,b6,b7 : out bit_vector(15 downto 0);
      CLK : bit);
end reg_h;
architecture beh of reg_h is
begin
process
variable d0,d1,d2,d3,d4,d5,d6,d7 : bit_vector(15 downto 0);
begin
d0 := a0;
d1 := a1;
d2 := a2;
d3 := a3;
d4 := a4;
d5 := a5;
d6 := a6;
d7 := a7;
wait until CLK'event and CLK = '1';
b0 <= d0;

```

```

    b1 <= d1;
    b2 <= d2;
    b3 <= d3;
    b4 <= d4;
    b5 <= d5;
    b6 <= d6;
    b7 <= d7;
    wait on CLK;
    end process;
end beh;
----- Adder_i -----
use work.pack1.all;
entity add_i is
    port(a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13,a14,a15,a16:
        bit_vector(15 downto 0);
        b1,b2,b3,b4,b5,b6,b7,b8 : out bit_vector(15 downto 0);
        CLK : bit);
end add_i;
architecture beh of add_i is
    begin
        process
            variable x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,x13,x14,x15,x16,
                n1,n2,n3,n4,n5,n6,n7,n8 : bit_vector(15 downto 0);
            variable y1,y2,y3,y4,y5,y6,y7,y8,y9,y10,y11,y12,y13,y14,y15,y16,
                m1,m2,m3,m4,m5,m6,m7,m8 : integer := 0;
        begin
            x1 := a1; x2 := a2; x3 := a3; x4 := a4;
            x5 := a5; x6 := a6; x7 := a7; x8 := a8;
            x9 := a9; x10 := a10; x11 := a11; x12 := a12;
            x13 := a13; x14 := a14; x15 := a15; x16 := a16;
            bi_to_in(x1,y1);bi_to_in(x2,y2);bi_to_in(x3,y3);bi_to_in(x4,y4);
            bi_to_in(x5,y5);bi_to_in(x6,y6);bi_to_in(x7,y7);bi_to_in(x8,y8);
            bi_to_in(x9,y9);bi_to_in(x10,y10);bi_to_in(x11,y11);
            bi_to_in(x12,y12);
            bi_to_in(x13,y13);bi_to_in(x14,y14);bi_to_in(x15,y15);
            bi_to_in(x16,y16);
            m1 := y1 + y2; m2 := y3 + y4; m3 := y5 + y6; m4 := y7 + y8;
            m5 := y9 + y10; m6 := y11 + y12; m7 := y13 + y14; m8 := y15 + y16;

            in_to_bi(m1,n1); in_to_bi(m2,n2); in_to_bi(m3,n3); in_to_bi(m4,n4);
            in_to_bi(m5,n5); in_to_bi(m6,n6); in_to_bi(m7,n7); in_to_bi(m8,n8);

            b1 <= n1; b2 <= n2; b3 <= n3; b4 <= n4;

```

```

    b5 <= n5; b6 <= n6; b7 <= n7; b8 <= n8;

    wait on a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13,a14,a15,a16;
end process;
end beh;
----- Shift right 2-bit register -----
entity shi_2 is
    port(a1,a2,a3,a4,a5,a6,a7,a8 : bit_vector(15 downto 0);
          sr1,sr2,sr3,sr4,sr5,sr6,sr7,sr8,b1,b2,b3,b4,b5,b6,b7,b8 :
          out bit_vector( 15 downto 0);clr : bit_vector(15 downto 0);
          CLK : bit);
end shi_2;
architecture beh of shi_2 is
    begin
        process
            variable x1,x2,x3,x4,x5,x6,x7,x8,y1,y2,y3,y4,y5,y6,y7,y8 :
                bit_vector(15 downto 0);
            variable i : integer := 0;
            begin
                wait until CLK'event and CLK = '1';
                x1 := a1; x2 := a2; x3 := a3; x4 := a4;
                x5 := a5; x6 := a6; x7 := a7; x8 := a8;
                if x1(15)='0' then
                    y1(13) := x1(15); y1(12) := x1(14); y1(11) := x1(13);
                    y1(10) := x1(12); y1(9) := X1(11); Y1(8) := X1(10);
                    y1(7) := X1(9); y1(6) := x1(8); y1(5) := x1(7);
                    y1(4) := x1(6); y1(3) := x1(5); y1(2) := x1(4);
                    y1(1) := x1(3); y1(0) := x1(2); y1(14) := '0';
                    y1(15) := '0';
                else
                    y1(13) := x1(15); y1(12) := x1(14); y1(11) := x1(13);
                    y1(10) := x1(12); y1(9) := X1(11); Y1(8) := X1(10);
                    y1(7) := X1(9); y1(6) := x1(8); y1(5) := x1(7);
                    y1(4) := x1(6); y1(3) := x1(5); y1(2) := x1(4);
                    y1(1) := x1(3); y1(0) := x1(2); y1(14) := '1';
                    y1(15) := '1';
                end if;
            end if;
        end process
    end beh;
    -----
    if x2(15)='0' then
        y2(13) := x2(15); y2(12) := x2(14); y2(11) := x2(13);
        y2(10) := x2(12); y2(9) := X2(11); Y2(8) := X2(10);
        y2(7) := X2(9); y2(6) := x2(8); y2(5) := x2(7);
        y2(4) := x2(6); y2(3) := x2(5); y2(2) := x2(4);
    end if;

```

```

y2(1) := x2(3); y2(0) := x2(2); y2(14) := '0';
y2(15) := '0';
else
y2(13) := x2(15); y2(12) := x2(14); y2(11) := x2(13);
y2(10) := x2(12); y2(9) := X2(11); Y2(8) := X2(10);
y2(7) := X2(9); y2(6) := x2(8); y2(5) := x2(7);
y2(4) := x2(6); y2(3) := x2(5); y2(2) := x2(4);
y2(1) := x2(3); y2(0) := x2(2); y2(14) := '1';
y2(15) := '1';
end if;

```

```

if x3(15)='0' then
y3(13) := x3(15); y3(12) := x3(14); y3(11) := x3(13);
y3(10) := x3(12); y3(9) := X3(11); y3(8) := x3(10);
y3(7) := X3(9); y3(6) := x3(8); y3(5) := x3(7);
y3(4) := x3(6); y3(3) := x3(5); y3(2) := x3(4);
y3(1) := x3(3); y3(0) := x3(2); y3(14) := '0';
y3(15) := '0';
else
y3(13) := x3(15); y3(12) := x3(14); y3(11) := x3(13);
y3(10) := x3(12); y3(9) := X3(11); Y3(8) := X3(10);
y3(7) := X3(9); y3(6) := x3(8); y3(5) := x3(7);
y3(4) := x3(6); y3(3) := x3(5); y3(2) := x3(4);
y3(1) := x3(3); y3(0) := x3(2); y3(14) := '1';
y3(15) := '1';
end if;

```

```

if x4(15)='0' then
y4(13) := x4(15); y4(12) := x4(14); y4(11) := x4(13);
y4(10) := x4(12); y4(9) := X4(11); y4(8) := x4(10);
y4(7) := X4(9); y4(6) := x4(8); y4(5) := x4(7);
y4(4) := x4(6); y4(3) := x4(5); y4(2) := x4(4);
y4(1) := x4(3); y4(0) := x4(2); y4(14) := '0';
y4(15) := '0';
else
y4(13) := x4(15); y4(12) := x4(14); y4(11) := x4(13);
y4(10) := x4(12); y4(9) := X4(11); Y4(8) := X4(10);
y4(7) := X4(9); y4(6) := x4(8); y4(5) := x4(7);
y4(4) := x4(6); y4(3) := x4(5); y4(2) := x4(4);
y4(1) := x4(3); y4(0) := x4(2); y4(14) := '1';
y4(15) := '1';
end if;

```

```

if x5(15)='0' then
  y5(13) := x5(15); y5(12) := x5(14); y5(11) := x5(13);
  y5(10) := x5(12); y5(9) := X5(11); y5(8) := x5(10);
  y5(7) := X5(9); y5(6) := x5(8); y5(5) := x5(7);
  y5(4) := x5(6); y5(3) := x5(5); y5(2) := x5(4);
  y5(1) := x5(3); y5(0) := x5(2); y5(14) := '0';
  y5(15) := '0';
else
  y5(13) := x5(15); y5(12) := x5(14); y5(11) := x5(13);
  y5(10) := x5(12); y5(9) := x5(11); y5(8) := x5(10);
  y5(7) := X5(9); y5(6) := x5(8); y5(5) := x5(7);
  y5(4) := x5(6); y5(3) := x5(5); y5(2) := x5(4);
  y5(1) := x5(3); y5(0) := x5(2); y5(14) := '1';
  y5(15) := '1';
end if;

```

```

if x6(15)='0' then
  y6(13) := x6(15); y6(12) := x6(14); y6(11) := x6(13);
  y6(10) := x6(12); y6(9) := X6(11); y6(8) := x6(10);
  y6(7) := X6(9); y6(6) := x6(8); y6(5) := x6(7);
  y6(4) := x6(6); y6(3) := x6(5); y6(2) := x6(4);
  y6(1) := x6(3); y6(0) := x6(2); y6(14) := '0';
  y6(15) := '0';
else
  y6(13) := x6(15); y6(12) := x6(14); y6(11) := x6(13);
  y6(10) := x6(12); y6(9) := x6(11); y6(8) := x6(10);
  y6(7) := X6(9); y6(6) := x6(8); y6(5) := x6(7);
  y6(4) := x6(6); y6(3) := x6(5); y6(2) := x6(4);
  y6(1) := x6(3); y6(0) := x6(2); y6(14) := '1';
  y6(15) := '1';
end if;

```

```

if x7(15)='0' then
  y7(13) := x7(15); y7(12) := x7(14); y7(11) := x7(13);
  y7(10) := x7(12); y7(9) := X7(11); y7(8) := x7(10);
  y7(7) := X7(9); y7(6) := x7(8); y7(5) := x7(7);
  y7(4) := x7(6); y7(3) := x7(5); y7(2) := x7(4);
  y7(1) := x7(3); y7(0) := x7(2); y7(14) := '0';
  y7(15) := '0';
else
  y7(13) := x7(15); y7(12) := x7(14); y7(11) := x7(13);
  y7(10) := x7(12); y7(9) := x7(11); y7(8) := x7(10);
  y7(7) := X7(9); y7(6) := x7(8); y7(5) := x7(7);

```

```

    y7(4) := x7(6); y7(3) := x7(5); y7(2) := x7(4);
    y7(1) := x7(3); y7(0) := x7(2); y7(14) := '1';
    y7(15) := '1';
end if;
-----
    if x8(15)='0' then
        y8(13) := x8(15); y8(12) := x8(14); y8(11) := x8(13);
        y8(10) := x8(12); y8(9) := x8(11); y8(8) := x8(10);
        y8(7) := x8(9); y8(6) := x8(8); y8(5) := x8(7);
        y8(4) := x8(6); y8(3) := x8(5); y8(2) := x8(4);
        y8(1) := x8(3); y8(0) := x8(2); y8(14) := '0';
        y8(15) := '0';
    else
        y8(13) := x8(15); y8(12) := x8(14); y8(11) := x8(13);
        y8(10) := x8(12); y8(9) := x8(11); y8(8) := x8(10);
        y8(7) := x8(9); y8(6) := x8(8); y8(5) := x8(7);
        y8(4) := x8(6); y8(3) := x8(5); y8(2) := x8(4);
        y8(1) := x8(3); y8(0) := x8(2); y8(14) := '1';
        y8(15) := '1';
    end if;
-----
    sr1 <= y1; sr2 <= y2; sr3 <= y3; sr4 <= y4;
    sr5 <= y5; sr6 <= y6; sr7 <= y7; sr8 <= y8;
    i := i+1;
    if i = 6 then
        b1 <= y1; b2 <= y2; b3 <= y3; b4 <= y4;
        b5 <= y5; b6 <= y6; b7 <= y7; b8 <= y8;
        x1 := clr; x2 := clr; x3 := clr; x4 := clr;
        x5 := clr; x6 := clr; x7 := clr; x8 := clr;
        sr1 <= clr; sr2 <= clr; sr3 <= clr; sr4 <= clr;
        sr5 <= clr; sr6 <= clr; sr7 <= clr; sr8 <= clr;
        i := 0;
    end if;
    wait on a1,a2,a3,a4,a5,a6,a7,a8,clr,CLK;
end process;
end beh;
----- Result output -----
entity result is
    port(a1,a2,a3,a4,a5,a6,a7,a8 : bit_vector(15 downto 0);
          k : out bit_vector(15 downto 0);CLK : bit);
end result;
architecture beh of result is
    type r is array(0 to 7) of bit_vector(15 downto 0);

```



```

begin
  process
    variable x : r;
    begin
      x(0) := a1; x(1) := a2; x(2) := a3; x(3) := a4;
      x(4) := a5; x(5) := a6; x(6) := a7; x(7) := a8;
      for i in 0 to 7 loop
        wait until CLK'event and CLK = '1';
        k <= x(i);
      end loop;
      wait on a1,a2,a3,a4,a5,a6,a7,a8,CLK;
    end process;
end beh;

----- Test bench -----
use work.pack1.all;
entity test is end test;
architecture str of test is
  component clock_ge port(CLCK :inout bit);
  end component;
  component clock port(CLK :inout bit);
  end component;
  component control port(CLK : bit;ct : out bit);
  end component;
  component LOAD port(AI : in bit_vector(11 downto 0);
    B0,B1,B2,B3,B4,B5,B6,B7 : out bit_vector(11 downto 0);
    CLK : in bit);
  end component;
  component shift
    port(bi0,bi1,bi2,bi3,bi4,bi5,bi6,bi7 : in bit_vector(11 downto 0);
      bo0,bo1,bo2,bo3,bo4,bo5,bo6,bo7 : out bit_vector(11 downto 0);
      CLK : in bit);
  end component;
  component adsu
    port(a0,a1,a2,a3,a4,a5,a6,a7 : bit_vector(1 downto 0);
      b0,b1,b2,b3,b4,b5,b6,b7 : out bit_vector(1 downto 0);
      CLK,cr,st : bit);
  end component;
  component reg
    port(a0,a1,a2,a3,a4,a5,a6,a7 : bit_vector(1 downto 0);
      b0,b1,b2,b3,b4,b5,b6,b7 : out bit_vector(1 downto 0);
      CLK : bit);
  end component;
  component rom

```

```

    port(e0,e1,e2,e3,e4,e5,e6,e7 : bit_vector(1 downto 0);
          b10,b11,b20,b21,b30,b31,b40,b41,b50,b51,b60,b61,b70,b71,b80,b81:
          out bit_vector(15 downto 0);
          CLK : bit);
end component;
component shi_1
    port(f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,f13,f14,f15,f16:
          bit_vector(15 downto 0);
          b10,b11,b20,b21,b30,b31,b40,b41,b50,b51,b60,b61,b70,b71,b80,b81:
          out bit_vector(15 downto 0);
          CLK : bit);
end component;
component delay1
    port(a: bit;b: out bit;CLK: bit);
end component;
component delay2
    port(a: bit;b: out bit;CLK: bit);
end component;
component delay3
    port(a: bit;b: out bit;CLK: bit);
end component;
component delay4
    port(a: bit;b: out bit;CLK: bit);
end component;
component delay5
    port(a: bit;b: out bit;CLK: bit);
end component;
component delay6
    port(a: bit;b: out bit;CLK: bit);
end component;
component delay7
    port(a: bit;b: out bit;CLK: bit);
end component;
component delay8
    port(a: bit;b: out bit;CLK: bit);
end component;
component delay9
    port(a: bit;b: out bit;CLK: bit);
end component;
component delay10
    port(a: bit;b: out bit;CLK: bit);
end component;
component add_g

```

```

    port(a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13,a14,a15,a16:
        bit_vector(15 downto 0);
        b1,b2,b3,b4,b5,b6,b7,b8 : out bit_vector(15 downto 0);
        CLK,as : bit);
end component;
component reg_h
    port(a0,a1,a2,a3,a4,a5,a6,a7 : bit_vector(15 downto 0);
        b0,b1,b2,b3,b4,b5,b6,b7 : out bit_vector(15 downto 0);
        CLK : bit);
end component;
component add_i
    port(a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13,a14,a15,a16:
        bit_vector(15 downto 0);b1,b2,b3,b4,b5,b6,b7,b8 :
        out bit_vector(15 downto 0);CLK : bit);
end component;
component shi_2
    port(a1,a2,a3,a4,a5,a6,a7,a8 : bit_vector(15 downto 0);
        sr1,sr2,sr3,sr4,sr5,sr6,sr7,sr8,b1,b2,b3,b4,b5,b6,b7,b8 :
        out bit_vector(15 downto 0);clr : bit_vector(15 downto 0);
        CLK : bit);
end component;
component result
    port(a1,a2,a3,a4,a5,a6,a7,a8 : bit_vector(15 downto 0);
        k : out bit_vector(15 downto 0); CLK : bit );
end component;
for C: clock_ge use entity work.clock_ge(clk_ctl);
for ad: clock use entity work.clock(beh);
for a : control use entity work.control(beh);
for L : LOAD use entity work.LOAD(BEH);
for S : shift use entity work.shift(beh);
for D : adsu use entity work.adsu(beh);
for r : reg use entity work.reg(beh);
for o : rom use entity work.rom(beh);
for s_1 : shi_1 use entity work.shi_1(beh);
for b : delay1 use entity work.delay1(beh);
for e : delay2 use entity work.delay2(beh);
for dely3 : delay3 use entity work.delay3(beh);
for dely4 : delay4 use entity work.delay4(beh);
for dely5 : delay5 use entity work.delay5(beh);
for dely6 : delay6 use entity work.delay6(beh);
for dely7 : delay7 use entity work.delay7(beh);
for dely8 : delay8 use entity work.delay8(beh);
for dely9 : delay9 use entity work.delay9(beh);

```

```

for dely10 : delay10 use entity work.delay10(beh);
for g : add_g use entity work.add_g(beh);
for h : reg_h use entity work.reg_h(beh);
for i : add_i use entity work.add_i(beh);
for j : shi_2 use entity work.shi_2(beh);
for t : result use entity work.result(beh);
signal di : bit_vector(11 downto 0);
signal ck : bit;
signal clk : bit;
signal go : bit;
signal io : bit;
signal ho : bit;
signal te : bit;
signal de : bit;
signal ab : bit;
signal cd : bit;
signal ef : bit;
signal gh : bit;
signal ij : bit;
signal kl : bit;
signal d0,d1,d2,d3,d4,d5,d6,d7 : bit_vector(11 downto 0);
Signal so0,so1,so2,so3,so4,so5,so6,so7 : bit_vector(1 downto 0);
signal co0,co1,co2,co3,co4,co5,co6,co7 : bit_vector(1 downto 0);
signal do0,do1,do2,do3,do4,do5,do6,do7 : bit_vector(1 downto 0);
signal clr : bit := '0';
signal set : bit := '0';
signal e1,e2,e3,e4,e5,e6,e7,e8,e9,e10,e11,e12,e13,e14,e15,e16 :
    bit_vector(15 downto 0);
signal f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,f13,f14,f15,f16:
    bit_vector(15 downto 0);
signal g1,g2,g3,g4,g5,g6,g7,g8 : bit_vector(15 downto 0);
    signal h1,h2,h3,h4,h5,h6,h7,h8 : bit_vector(15 downto 0);
signal i1,i2,i3,i4,i5,i6,i7,i8 : bit_vector(15 downto 0);
signal j1,j2,j3,j4,j5,j6,j7,j8 : bit_vector(15 downto 0);
    signal r1,r2,r3,r4,r5,r6,r7,r8 : bit_vector(15 downto 0);
signal cr : bit_vector(15 downto 0) := "0000000000000000";
signal p : bit_vector(15 downto 0);
begin
    C : clock_ge port map(ck);
    ad : clock port map(clck);
    a : control port map(ck,go);
    b : delay1 port map(go,io,ck);
    e : delay2 port map(ck,ho,clck);

```

```

dely3 : delay3 port map(ho,te,clck);
dely4 : delay4 port map(te,de,clck);
dely5 : delay5 port map(de,ab,clck);
dely6 : delay6 port map(ab,cd,clck);
dely7 : delay7 port map(cd,ef,clck);
dely8 : delay8 port map(ef,gh,clck);
dely9 : delay9 port map(gh,ij,clck);
dely10 : delay10 port map(ij,kl,clck);
L : LOAD port map(di,d0,d1,d2,d3,d4,d5,d6,d7,ck);
S : shift port map(d0,d1,d2,d3,d4,d5,d6,d7,
    so0,so1,so2,so3,so4,so5,so6,so7,ck);
D : adsu port map(so0,so1,so2,so3,so4,so5,so6,so7,
    co0,co1,co2,co3,co4,co5,co6,co7,
    ck,clr,set);
r : reg port map(co0,co1,co2,co3,co4,co5,co6,co7,
    do0,do1,do2,do3,do4,do5,do6,do7,
    ck);
o : rom port map(do0,do1,do2,do3,do4,do5,do6,do7,
    e1,e2,e3,e4,e5,e6,e7,e8,e9,e10,e11,e12,e13,e14,
    e15,e16,ck);
s_1 : shi_1 port map(e1,e2,e3,e4,e5,e6,e7,e8,e9,e10,e11,e12,e13,e14,e15,e16,
    f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,f13,f14,f15,f16,
    ck);
g : add_g port map(f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,f13,f14,f15,f16,
    g1,g2,g3,g4,g5,g6,g7,g8,ck,io);
h : reg_h port map(g1,g2,g3,g4,g5,g6,g7,g8,h1,h2,h3,h4,h5,h6,h7,h8,ck);

i : add_i port map(h1,r1,h2,r2,h3,r3,h4,r4,h5,r5,h6,r6,h7,r7,h8,r8,
    i1,i2,i3,i4,i5,i6,i7,i8,ck);
j : shi_2 port map(i1,i2,i3,i4,i5,i6,i7,i8,r1,r2,r3,r4,r5,r6,r7,r8,
    j1,j2,j3,j4,j5,j6,j7,j8,cr,kl);
t : result port map(j1,j2,j3,j4,j5,j6,j7,j8,p,ck);
set <= '1' after 5 ns;
di <= "000101101010" after 7 ns,
    "000000000000" after 17 ns,
    "000101101010" after 27 ns,
    "001011010100" after 37 ns,
    "000101101010" after 47 ns,
    "000000000000" after 57 ns,
    "000101101010" after 67 ns,
    "001011010100" after 77 ns;
end str;

```

APPENDIX B. 16-BIT 1-D DCT VHDL SOURCE CODE

```

----- Shift right 2-bit register -----
entity shi_2 is
  port(a1,a2,a3,a4,a5,a6,a7,a8 : bit_vector(15 downto 0);
        sr1,sr2,sr3,sr4,sr5,sr6,sr7,sr8,b1,b2,b3,b4,b5,b6,b7,b8 :
        out bit_vector( 15 downto 0);clr : bit_vector(15 downto 0);
        CLK : bit);
end shi_2;
architecture beh of shi_2 is
  begin
    process
      variable x1,x2,x3,x4,x5,x6,x7,x8,y1,y2,y3,y4,y5,y6,y7,y8 :
        bit_vector(15 downto 0);
      variable i : integer := 0;
    begin
      wait until CLK'event and CLK = '1';
      x1 := a1; x2 := a2; x3 := a3; x4 := a4;
      x5 := a5; x6 := a6; x7 := a7; x8 := a8;
      if x1(15)='0' then
        y1(13) := x1(15); y1(12) := x1(14); y1(11) := x1(13);
        y1(10) := x1(12); y1(9) := X1(11); Y1(8) := X1(10);
        y1(7) := X1(9); y1(6) := x1(8); y1(5) := x1(7);
        y1(4) := x1(6); y1(3) := x1(5); y1(2) := x1(4);
        y1(1) := x1(3); y1(0) := x1(2); y1(14) := '0';
        y1(15) := '0';
      else
        y1(13) := x1(15); y1(12) := x1(14); y1(11) := x1(13);
        y1(10) := x1(12); y1(9) := X1(11); Y1(8) := X1(10);
        y1(7) := X1(9); y1(6) := x1(8); y1(5) := x1(7);
        y1(4) := x1(6); y1(3) := x1(5); y1(2) := x1(4);
        y1(1) := x1(3); y1(0) := x1(2); y1(14) := '1';
        y1(15) := '1';
      end if;
    end process;

    -----
    if x2(15)='0' then
      y2(13) := x2(15); y2(12) := x2(14); y2(11) := x2(13);
      y2(10) := x2(12); y2(9) := X2(11); Y2(8) := X2(10);
      y2(7) := X2(9); y2(6) := x2(8); y2(5) := x2(7);
      y2(4) := x2(6); y2(3) := x2(5); y2(2) := x2(4);
    end if;
  end architecture;

```

```

y2(1) := x2(3); y2(0) := x2(2); y2(14) := '0';
y2(15) := '0';
else
y2(13) := x2(15); y2(12) := x2(14); y2(11) := x2(13);
y2(10) := x2(12); y2(9) := X2(11); Y2(8) := X2(10);
y2(7) := X2(9); y2(6) := x2(8); y2(5) := x2(7);
y2(4) := x2(6); y2(3) := x2(5); y2(2) := x2(4);
y2(1) := x2(3); y2(0) := x2(2); y2(14) := '1';
y2(15) := '1';
end if;

```

```

if x3(15)='0' then
y3(13) := x3(15); y3(12) := x3(14); y3(11) := x3(13);
y3(10) := x3(12); y3(9) := X3(11); y3(8) := x3(10);
y3(7) := X3(9); y3(6) := x3(8); y3(5) := x3(7);
y3(4) := x3(6); y3(3) := x3(5); y3(2) := x3(4);
y3(1) := x3(3); y3(0) := x3(2); y3(14) := '0';
y3(15) := '0';
else
y3(13) := x3(15); y3(12) := x3(14); y3(11) := x3(13);
y3(10) := x3(12); y3(9) := X3(11); Y3(8) := X3(10);
y3(7) := X3(9); y3(6) := x3(8); y3(5) := x3(7);
y3(4) := x3(6); y3(3) := x3(5); y3(2) := x3(4);
y3(1) := x3(3); y3(0) := x3(2); y3(14) := '1';
y3(15) := '1';
end if;

```

```

if x4(15)='0' then
y4(13) := x4(15); y4(12) := x4(14); y4(11) := x4(13);
y4(10) := x4(12); y4(9) := X4(11); y4(8) := x4(10);
y4(7) := X4(9); y4(6) := x4(8); y4(5) := x4(7);
y4(4) := x4(6); y4(3) := x4(5); y4(2) := x4(4);
y4(1) := x4(3); y4(0) := x4(2); y4(14) := '0';
y4(15) := '0';
else
y4(13) := x4(15); y4(12) := x4(14); y4(11) := x4(13);
y4(10) := x4(12); y4(9) := X4(11); Y4(8) := X4(10);
y4(7) := X4(9); y4(6) := x4(8); y4(5) := x4(7);
y4(4) := x4(6); y4(3) := x4(5); y4(2) := x4(4);
y4(1) := x4(3); y4(0) := x4(2); y4(14) := '1';
y4(15) := '1';
end if;

```

```

if x5(15)='0' then
  y5(13) := x5(15); y5(12) := x5(14); y5(11) := x5(13);
  y5(10) := x5(12); y5(9) := X5(11); y5(8) := x5(10);
  y5(7) := X5(9); y5(6) := x5(8); y5(5) := x5(7);
  y5(4) := x5(6); y5(3) := x5(5); y5(2) := x5(4);
  y5(1) := x5(3); y5(0) := x5(2); y5(14) := '0';
  y5(15) := '0';
else
  y5(13) := x5(15); y5(12) := x5(14); y5(11) := x5(13);
  y5(10) := x5(12); y5(9) := x5(11); y5(8) := x5(10);
  y5(7) := X5(9); y5(6) := x5(8); y5(5) := x5(7);
  y5(4) := x5(6); y5(3) := x5(5); y5(2) := x5(4);
  y5(1) := x5(3); y5(0) := x5(2); y5(14) := '1';
  y5(15) := '1';
end if;

```

```

if x6(15)='0' then
  y6(13) := x6(15); y6(12) := x6(14); y6(11) := x6(13);
  y6(10) := x6(12); y6(9) := X6(11); y6(8) := x6(10);
  y6(7) := X6(9); y6(6) := x6(8); y6(5) := x6(7);
  y6(4) := x6(6); y6(3) := x6(5); y6(2) := x6(4);
  y6(1) := x6(3); y6(0) := x6(2); y6(14) := '0';
  y6(15) := '0';
else
  y6(13) := x6(15); y6(12) := x6(14); y6(11) := x6(13);
  y6(10) := x6(12); y6(9) := x6(11); y6(8) := x6(10);
  y6(7) := X6(9); y6(6) := x6(8); y6(5) := x6(7);
  y6(4) := x6(6); y6(3) := x6(5); y6(2) := x6(4);
  y6(1) := x6(3); y6(0) := x6(2); y6(14) := '1';
  y6(15) := '1';
end if;

```

```

if x7(15)='0' then
  y7(13) := x7(15); y7(12) := x7(14); y7(11) := x7(13);
  y7(10) := x7(12); y7(9) := X7(11); y7(8) := x7(10);
  y7(7) := X7(9); y7(6) := x7(8); y7(5) := x7(7);
  y7(4) := x7(6); y7(3) := x7(5); y7(2) := x7(4);
  y7(1) := x7(3); y7(0) := x7(2); y7(14) := '0';
  y7(15) := '0';
else
  y7(13) := x7(15); y7(12) := x7(14); y7(11) := x7(13);
  y7(10) := x7(12); y7(9) := x7(11); y7(8) := x7(10);
  y7(7) := X7(9); y7(6) := x7(8); y7(5) := x7(7);

```



```

y7(4) := x7(6); y7(3) := x7(5); y7(2) := x7(4);
y7(1) := x7(3); y7(0) := x7(2); y7(14) := '1';
y7(15) := '1';
end if;

```

```

if x8(15)='0' then
  y8(13) := x8(15); y8(12) := x8(14); y8(11) := x8(13);
  y8(10) := x8(12); y8(9) := x8(11); y8(8) := x8(10);
  y8(7) := x8(9); y8(6) := x8(8); y8(5) := x8(7);
  y8(4) := x8(6); y8(3) := x8(5); y8(2) := x8(4);
  y8(1) := x8(3); y8(0) := x8(2); y8(14) := '0';
  y8(15) := '0';
else
  y8(13) := x8(15); y8(12) := x8(14); y8(11) := x8(13);
  y8(10) := x8(12); y8(9) := x8(11); y8(8) := x8(10);
  y8(7) := x8(9); y8(6) := x8(8); y8(5) := x8(7);
  y8(4) := x8(6); y8(3) := x8(5); y8(2) := x8(4);
  y8(1) := x8(3); y8(0) := x8(2); y8(14) := '1';
  y8(15) := '1';
end if;

```

```

sr1 <= y1; sr2 <= y2; sr3 <= y3; sr4 <= y4;
sr5 <= y5; sr6 <= y6; sr7 <= y7; sr8 <= y8;
i := i+1;
if i = 8 then
  b1 <= y1; b2 <= y2; b3 <= y3; b4 <= y4;
  b5 <= y5; b6 <= y6; b7 <= y7; b8 <= y8;
  x1 := clr; x2 := clr; x3 := clr; x4 := clr;
  x5 := clr; x6 := clr; x7 := clr; x8 := clr;
  sr1 <= clr; sr2 <= clr; sr3 <= clr; sr4 <= clr;
  sr5 <= clr; sr6 <= clr; sr7 <= clr; sr8 <= clr;
  i := 0;
end if;

```

```

wait on a1,a2,a3,a4,a5,a6,a7,a8,clr,CLK;
end process;

```

```

end beh;

```

```

----- Test bench -----

```

```

use work.pack1.all;
entity test is end test;
architecture str of test is
  component clock_ge port(CLCK :inout bit);
  end component;
  component clock port(CLK :inout bit);

```

```

end component;

component control port(CLK : bit;ct : out bit);
end component;
component LOAD port(AI : in bit_vector(15 downto 0);
                    B0,B1,B2,B3,B4,B5,B6,B7 : out bit_vector(15 downto 0);
                    CLK : in bit);
end component;
component shift
    port(bi0,bi1,bi2,bi3,bi4,bi5,bi6,bi7 : in bit_vector(15 downto 0);
          bo0,bo1,bo2,bo3,bo4,bo5,bo6,bo7 : out bit_vector(1 downto 0);
          CLK : in bit);
end component;
component adsu
    port(a0,a1,a2,a3,a4,a5,a6,a7 : bit_vector(1 downto 0);
          b0,b1,b2,b3,b4,b5,b6,b7 : out bit_vector(1 downto 0);
          CLK,cr,st : bit);
end component;
component reg
    port(a0,a1,a2,a3,a4,a5,a6,a7 : bit_vector(1 downto 0);
          b0,b1,b2,b3,b4,b5,b6,b7 : out bit_vector(1 downto 0);
          CLK : bit);
end component;
component rom
    port(e0,e1,e2,e3,e4,e5,e6,e7 : bit_vector(1 downto 0);
          b10,b11,b20,b21,b30,b31,b40,b41,b50,b51,b60,b61,b70,b71,b80,b81 :
          out bit_vector(15 downto 0);
          CLK : bit);
end component;
component shi_1
    port(f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,f13,f14,f15,f16 :
          bit_vector(15 downto 0);
          b10,b11,b20,b21,b30,b31,b40,b41,b50,b51,b60,b61,b70,b71,b80,b81 :
          out bit_vector(15 downto 0);
          CLK : bit);
end component;
component delay1
    port(a: bit;b: out bit;CLK: bit);
end component;
component delay2
    port(a: bit;b: out bit;CLK: bit);
end component;
component delay3

```

```

    port(a: bit;b: out bit;CLK: bit);
end component;

component delay4
    port(a: bit;b: out bit;CLK: bit);
end component;
component delay15
    port(a: bit;b: out bit;CLK: bit);
end component;
component delay16
    port(a: bit;b: out bit;CLK: bit);
end component;
component delay17
    port(a: bit;b: out bit;CLK: bit);
end component;
component delay18
    port(a: bit;b: out bit;CLK: bit);
end component;
component add_g
    port(a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13,a14,a15,a16:
        bit_vector(15 downto 0);
        b1,b2,b3,b4,b5,b6,b7,b8 : out bit_vector(15 downto 0);
        CLK,as : bit);
end component;
component reg_h
    port(a0,a1,a2,a3,a4,a5,a6,a7 : bit_vector(15 downto 0);
        b0,b1,b2,b3,b4,b5,b6,b7 : out bit_vector(15 downto 0);
        CLK : bit);
end component;
component add_i
    port(a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13,a14,a15,a16:
        bit_vector(15 downto 0);b1,b2,b3,b4,b5,b6,b7,b8 :
        out bit_vector(15 downto 0);CLK : bit);
end component;
component shi_2
    port(a1,a2,a3,a4,a5,a6,a7,a8 : bit_vector(15 downto 0);
        sr1,sr2,sr3,sr4,sr5,sr6,sr7,sr8,b1,b2,b3,b4,b5,b6,b7,b8 :
        out bit_vector(15 downto 0);clr : bit_vector(15 downto 0);
        CLK : bit);
end component;
component result
    port(a1,a2,a3,a4,a5,a6,a7,a8 : bit_vector(15 downto 0);
        k : out bit_vector(15 downto 0); CLK : bit );

```

```

end component;
for C: clock_ge use entity work.clock_ge(clk_ctl);
for ad: clock use entity work.clock(beh);
for a : control use entity work.control(beh);
for L : LOAD use entity work.LOAD(BEH);
for S : shift use entity work.shift(beh);
for D : adsu use entity work.adsu(beh);
for r : reg use entity work.reg(beh);
for o : rom use entity work.rom(beh);
for s_1 : shi_1 use entity work.shi_1(beh);
for b : delay1 use entity work.delay1(beh);
for e : delay2 use entity work.delay2(beh);
for dely3 : delay3 use entity work.delay3(beh);
for dely4 : delay4 use entity work.delay4(beh);
for dely15 : delay15 use entity work.delay15(beh);
for dely16 : delay16 use entity work.delay16(beh);
for dely17 : delay17 use entity work.delay17(beh);
for dely18 : delay18 use entity work.delay18(beh);
for g : add_g use entity work.add_g(beh);
for h : reg_h use entity work.reg_h(beh);
for i : add_i use entity work.add_i(beh);
for j : shi_2 use entity work.shi_2(beh);
for t : result use entity work.result(beh);
signal di : bit_vector(15 downto 0);
signal ck : bit;
signal clck : bit;
signal go : bit;
signal io : bit;
signal ho : bit;
signal te : bit;
signal de : bit;
signal op,qr,st,eo,ko,mo,qo,ro,so,uo : bit;
signal d0,d1,d2,d3,d4,d5,d6,d7 : bit_vector(15 downto 0);
Signal so0,so1,so2,so3,so4,so5,so6,so7 : bit_vector(1 downto 0);
signal co0,co1,co2,co3,co4,co5,co6,co7 : bit_vector(1 downto 0);
signal do0,d01,d02,d03,d04,d05,d06,d07 : bit_vector(1 downto 0);
signal clr : bit := '0';
signal set : bit := '0';
signal e1,e2,e3,e4,e5,e6,e7,e8,e9,e10,e11,e12,e13,e14,e15,e16 :
    bit_vector(15 downto 0);
signal f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,f13,f14,f15,f16:
    bit_vector(15 downto 0);

```

```

signal g1,g2,g3,g4,g5,g6,g7,g8 : bit_vector(15 downto 0);
  signal h1,h2,h3,h4,h5,h6,h7,h8 : bit_vector(15 downto 0);
signal i1,i2,i3,i4,i5,i6,i7,i8 : bit_vector(15 downto 0);
signal j1,j2,j3,j4,j5,j6,j7,j8 : bit_vector(15 downto 0);
  signal r1,r2,r3,r4,r5,r6,r7,r8 : bit_vector(15 downto 0);
signal cr : bit_vector(15 downto 0) := "0000000000000000";
signal p : bit_vector(15 downto 0);
begin
  C : clock_ge port map(ck);
  ad : clock port map(clck);
  a : control port map(ck,go);
  b : delay1 port map(go,io,ck);
  e : delay2 port map(ck,ho,clck);
  dely3 : delay3 port map(ho,te,clck);
  dely4 : delay4 port map(te,de,clck);
  dely15 : delay15 port map(io,eo,ck);
  dely16 : delay16 port map(eo,ko,ck);
  dely17 : delay17 port map(ko,mo,ck);
  dely18 : delay18 port map(mo,qo,ck);
  L : LOAD port map(di,d0,d1,d2,d3,d4,d5,d6,d7,ck);
  S : shift port map(d0,d1,d2,d3,d4,d5,d6,d7,
                    so0,so1,so2,so3,so4,so5,so6,so7,de);
  D : adsu port map(so0,so1,so2,so3,so4,so5,so6,so7,
                    co0,co1,co2,co3,co4,co5,co6,co7,
                    ck,clr,set);
  r : reg port map(co0,co1,co2,co3,co4,co5,co6,co7,
                    do0,do1,do2,do3,do4,do5,do6,do7,
                    ck);
  o : rom port map(do0,do1,do2,do3,do4,do5,do6,do7,
                    e1,e2,e3,e4,e5,e6,e7,e8,e9,e10,e11,e12,e13,e14,
                    e15,e16,ck);
s_1 : shi_1 port map(e1,e2,e3,e4,e5,e6,e7,e8,e9,e10,e11,e12,e13,e14,e15,e16,
                    f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,f13,f14,f15,f16,
                    ck);
  g : add_g port map(f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,f13,f14,f15,f16,
                    g1,g2,g3,g4,g5,g6,g7,g8,ck,qo);
  h : reg_h port map(g1,g2,g3,g4,g5,g6,g7,g8,h1,h2,h3,h4,h5,h6,h7,h8,ck);

  i : add_i port map(h1,r1,h2,r2,h3,r3,h4,r4,h5,r5,h6,r6,h7,r7,h8,r8,
                    i1,i2,i3,i4,i5,i6,i7,i8,ck);
  j : shi_2 port map(i1,i2,i3,i4,i5,i6,i7,i8,r1,r2,r3,r4,r5,r6,r7,r8,
                    j1,j2,j3,j4,j5,j6,j7,j8,cr,ho);
  t : result port map(j1,j2,j3,j4,j5,j6,j7,j8,p,ck);

```

```
set <= '1' after 5 ns;  
di <= "0000010110101000" after 7 ns,  
    "0000000000000000" after 17 ns,  
    "0000010110101000" after 27 ns,  
    "0000101101010000" after 37 ns,  
    "0000010110101000" after 47 ns,  
    "0000000000000000" after 57 ns,  
    "0000010110101000" after 67 ns,  
    "0000101101010000" after 77 ns;  
end str;
```

APPENDIX C. MATLAB PROGRAM OF DECIMAL-BINARY CONVERSION

```
while 1
    x(1,16) = 0;
    y = input('Please enter your number : ');
    if y == 0
        break
    end
    disp('wait!');
    if y > 0,
        x(1) = 0;
    else
        x(1) = 1;
        y = abs(y);
    end
    i = 2;
    for k = 1:15;
        if y > 1,
            x(i) = fix(y);
            y = y - x(i);
        else
            x(i) = 0;
        end
        y = 2 * y;
        i = i + 1;
    end
    disp(x);
end
```

APPENDIX D. STRUCTURAL 1-D DCT HAND CALCULATION

<p>①</p> <pre> 0000000000000000 + 0000000000000000 ----- 0000000000000000 </pre> <p>②</p> <pre> 0000000000000000 + 0101101010000010 ----- 0101101010000010 + 0000000000000000 ----- 0101101010000010 </pre> <p>③</p> <pre> 0010110101000001 + 0101101010000010 ----- 0111000100100010 + 0001011010100000 ----- 1000011111000010 </pre> <p>④</p> <pre> 0010110101000001 + 0101101010000010 ----- 0111000100100010 + 0010000111110000 ----- 1001001100010010 </pre>	<p>U0</p>	<p>⑤</p> <pre> 0010110101000001 + 0000000000000000 ----- 0001011010100000 + 0010010011000100 ----- 0011101101100100 0010110101000001 </pre> <p>⑥</p> <pre> 0000000000000000 + 0001011010100000 ----- 0000111011011001 + 0010010101111001 ----- 0010110101000001 </pre> <p>⑦</p> <pre> 0010110101000001 + 0000000000000000 ----- 0001011010100000 + 0000100101011110 ----- 0001111111111110 </pre> <p>⑧</p> <pre> 0000000000000000 - 0000000000000000 ----- 0000000000000000 + 0000011111111111 ----- 0000011111111111 </pre>
---	-----------	--

Fig. 19 U0 hand calculation

$$\begin{array}{r}
 \textcircled{1} \quad 0000000000000000 \\
 + \quad 0000000000000000 \\
 \hline
 0000000000000000 \\
 \textcircled{2} \quad 0000000000000000 \\
 + \quad 0101001000000011 \\
 \hline
 0101001000000011 \\
 + \quad 0000000000000000 \\
 \hline
 0101001000000011 \\
 \textcircled{3} \quad 000110000000101 \\
 + \quad 0011100111111101 \\
 \hline
 0100010111111111 \\
 + \quad 0001010010000000 \\
 \hline
 0101101001111111 \\
 \textcircled{4} \quad 000110000000101 \\
 + \quad 0011100111111101 \\
 \hline
 0100010111111111 \\
 + \quad 0001011010011111 \\
 \hline
 0101110010011110
 \end{array}$$

v1

$$\begin{array}{r}
 \textcircled{5} \quad 0011100111111101 \\
 + \quad 000110000000101 \\
 \hline
 0011010100000011 \\
 + \quad 0001011100100111 \\
 \hline
 0100110000101010 \\
 \textcircled{6} \quad 0011100111111101 \\
 + \quad 000110000000101 \\
 \hline
 0011010100000011 \\
 + \quad 0001001100001010 \\
 \hline
 0100100000001101 \\
 \textcircled{7} \quad 000110000000101 \\
 + \quad 000110000000101 \\
 \hline
 0010010000001111 \\
 + \quad 0001001000000011 \\
 \hline
 0011011000001010 \\
 \textcircled{8} \quad 000110000000101 \\
 - \quad 000110000000101 \\
 \hline
 1111001111111101 \\
 + \quad 0000110110000010 \\
 \hline
 0000000101111111
 \end{array}$$

Fig. 20 V1 hand calculation

①	0000000000000000		⑤	0001010001011101
+	0000000000000000		+	1100111011010111
	0000000000000000			1101100100000101
②	0000000000000000		+	1111111000111011
+	1110001100110100			1101011101000000
	1110001100110100		⑥	0001010001011101
+	0000000000000000		+	1100111011010111
	1110001100110100			1101100100000101
③	1110001100110100	V3	+	1111010111010000
	1100111011010111			1100111011010101
+	0001010001011101		⑦	1100111011010111
	1111101111001000		+	1100111011010111
+	1111100011001101			1011011001000010
	1111010010010101		+	1111001110110101
④	1100111011010111			1010100111110111
+	0001010001011101		⑧	1100111011010111
	1111101111001000		-	1100111011010111
+	1111110100100101			0001100010010100
	1111100011101101		+	1110101001111101
				0000001100010001

Fig. 21 V3 hand calculation

<p>① 0000000000000000 + 0000000000000000 0000000000000000</p> <p>② 0000000000000000 + 0000000000000000 0000000000000000</p> <p>③ 0010110101000001 + 0000000000000000 0001011010100000 + 0000000000000000 0001011010100000</p> <p>④ 0010110101000001 + 0000000000000000 0001011010100000 + 0000010110101000 0001110001001000</p>	U4	<p>⑤ 1101001010111111 + 0000000000000000 110100101011111 + 0000011100010010 1111000001110001</p> <p>⑥ 1101001010111111 + 0000000000000000 110100101011111 + 1111110000011100 1110010101111011</p> <p>⑦ 0010110101000001 + 0000000000000000 0001011010100000 + 1111100101011110 0000111111111110</p> <p>⑧ 0000000000000000 - 0000000000000000 0000000000000000 + 0000001111111111 0000001111111111</p>
---	----	---

Fig. 22 U4 hand calculation

①	0000000000000000		⑤	1111001001100101
+	0000000000000000		+	0010000011011001
	0000000000000000			0001101000001011
②	0000000000000000		+	0000000100101110
+	0001001100111110			0001101100111001
	0001001100111110		⑥	1111001001100101
+	0000000000000000		+	0010000011011001
	0001001100111110			0001101000001011
③	0010000011011001	V5	+	0000011011001110
+	1111001001100101			0010000011011001
	0000001011010001		⑦	0010000011011001
+	0000010011001111		+	0010000011011001
	0000011110100000			0011000101000101
④	0010000011011001		+	0000100000110110
+	1111001001100101			0011100101111011
	0000001011010001		⑧	0010000011011001
+	0000000111101000		-	0010000011011001
	0000010010111001			1110111110010011
			+	0000111001011110
				1111110111110001

Fig. 23 V5 hand calculation

①	0000000000000000	⑤	1111010001110111	
+	0000000000000000	+	111101100111001	
	0000000000000000		1111010101110100	
②	0000000000000000	+	111101101100100	
+	1110111110110000		1111000011011000	
	1110111110110000	⑥	1111010001110111	
+	0000000000000000	+	111101100111001	
	1110111110110000		1111010101110100	
③	1111101100111001	V7	+	1111110000110110
+	1111010001110111		1111000110101010	
	1111001000010011	⑦	111101100111001	
+	1111101111101100	+	111101100111001	
	1110110111111111		1111100011010101	
④	1111101100111001	+	1111110001101010	
+	1111010001110111		1111010100111111	
	1111001000010011	⑧	111101100111001	
+	1111101101111111	-	111101100111001	
	1110110110010010		0000001001100011	
		+	1111110101001111	
			1111111110110010	

Fig. 24 V7 hand calculation

APPENDIX E. FORMATION OF 2-BIT ADDER

A. TWO BIT ADDER TRUTH TABLE

Table XX Truth table of 2-bit adder

A_1	A_0	B_1	B_0	C_i	q_1	q_0	C_o
0	0	0	0	0	0	0	0
0	0	0	0	1	0	1	0
0	0	0	1	0	0	1	0
0	0	0	1	1	1	0	0
0	0	1	0	0	1	0	0
0	0	1	0	1	1	1	0
0	0	1	1	0	1	1	0
0	0	1	1	1	0	0	1
0	1	0	0	0	0	1	0
0	1	0	0	1	1	0	0
0	1	0	1	0	1	0	0
0	1	0	1	1	1	1	0
0	1	1	0	0	1	1	0
0	1	1	0	1	0	0	1
0	1	1	1	0	0	0	1
0	1	1	1	1	0	1	1

Table XXI (Table XX) continue

A_1	A_0	B_1	B_0	C_i	q_1	q_0	C_o
1	0	0	0	0	1	0	0
1	0	0	0	1	1	1	0
1	0	0	1	0	1	1	0
1	0	0	1	1	0	0	1
1	0	1	0	0	0	0	1
1	0	1	0	1	0	1	1
1	0	1	1	0	0	1	1
1	0	1	1	1	1	0	1
1	1	0	0	0	1	1	0
1	1	0	0	1	0	0	1
1	1	0	1	0	0	0	1
1	1	0	1	1	0	1	1
1	1	1	0	0	0	1	1
1	1	1	0	1	1	0	1
1	1	1	1	0	1	0	1
1	1	1	1	1	1	1	1

Two bit adder has five inputs, three outputs. A_1 , A_0 , B_1 , and B_0 represent the input and C_i represents the carrier in. Q_1 , q_0 represent the output and C_o represents the carrier out. After the set up of truth table, reduction can be made by Karnaugh map.

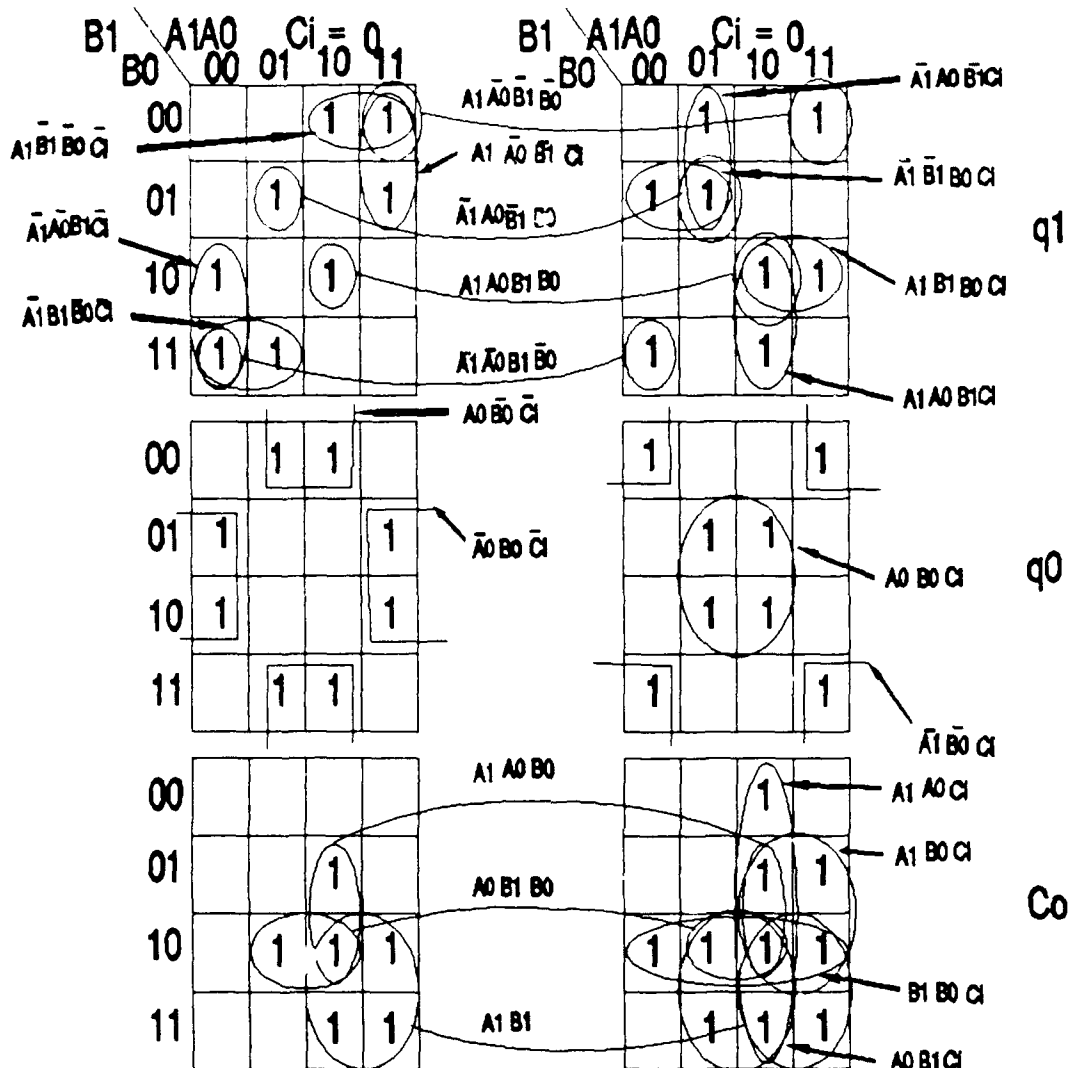


Fig. 25 Karnaugh map reduction

Karnaugh map reduction gives the reduced boolean expression.

$$\begin{aligned}
 q_1 = & \bar{A}_1 \bar{A}_0 B_1 \bar{C}_i + \bar{A}_1 B_1 \bar{B}_0 \bar{C}_i + A_1 \bar{B}_1 \bar{B}_0 \bar{C}_i + A_1 \bar{A}_0 \bar{B}_1 \bar{C}_i + \bar{A}_1 \bar{A}_0 \bar{B}_1 B_0 + A_1 \bar{A}_0 B_1 B_0 \\
 & + \bar{A}_1 \bar{A}_0 B_1 \bar{B}_0 + A_1 \bar{A}_0 \bar{B}_1 \bar{B}_0 + \bar{A}_1 \bar{A}_0 \bar{B}_1 C_i + \bar{A}_1 \bar{B}_1 B_0 C_i + A_1 B_1 B_0 C_i + A_1 \bar{A}_0 B_1 C_i
 \end{aligned} \quad (36)$$

$$q_0 = A_0 \bar{B}_0 \bar{C}_i + \bar{A}_0 B_0 \bar{C}_i + \bar{A}_0 \bar{B}_0 C_i + A_0 B_0 C_i \quad (37)$$

$$C_o = A_1 A_0 B_0 + A_0 B_1 B_0 + A_1 A_0 C_i + A_1 B_0 C_i + B_1 B_0 C_i + A_1 B_1 + A_0 B_1 C_i \quad (38)$$

LIST OF REFERENCES

1. L. J. D'Luna, *An 8×8 Discrete Cosine Transform Chip with Pixel Rate Clocks*, IEEE TH0303-8/90/0000, p7-5.1 - p7-5.4.
2. Herbert. Taub, *Digital Circuits and Microprocessors*, p58 - p81.
3. R. C. Gonzalez/P. Wintz, *Digital Image Processing*, p121 - p122.
4. Ernest. Meyer, *VHDL opens the road to Top-Down Design*, Computer Design, Feb. 1, 1989, p57 - p62.
5. Lipsett/Schaefer/Ussery, *VHDL : Hardware Description ana Design*, Kluwer Academic Publishers, 1989.
6. James R. Armstrong, *Chip-Level Modeling with VHDL*, Prentice Hall, 1989.
7. David L. Barton, *A First Course in VHDL*, Design Automation Guide, 1988.
8. *IEEE Standard VHDL Language Reference Manual Std 1076-1987*, Institute of Electronics Engineers, March 1988.
9. *386-MATLAB User's Guide*, The MathWorks, Inc.